University of Alberta Library
0 1620 2665752 6

Ex libris
universitatis
albertaensis

Digitized by the Internet Archive
in 2026 with funding from
University of Alberta Library

https://archive.org/details/0162026657526

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Christina Lau

TITLE OF THESIS: Object Management, Protection and Scheduling in FLEX

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: Spring 1988

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

THE UNIVERSITY OF ALBERTA

Object Management, Protection and Scheduling in FLEX

by

Christina Lau

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE
OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

SPRING, 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Object Management, Protection and Scheduling in FLEX** submitted by **Christina Lau** in partial fulfilment of the requirements for the degree of **Master of Science**.

Abstract

The FLEX project is an attempt to investigate the problems of integrating distributed database systems and distributed operating systems. In recent years, these issues have received considerable attention among database researchers. Most of this work has concentrated on the development of more suitable operating system algorithms to support distributed database management systems. FLEX is being implemented to serve as a testbed, concentrating on the architectural as well as the algorithmic issues. In this thesis, we discuss the initial design consideration and the global architecture of FLEX. We further describe, in detail, the design and implementation of the Object Manager, the Protection Mechanism and Scheduling in FLEX.

Acknowledgements

I wish to thank my supervisor Dr. Tamer Özsu for his patience and guidance and for providing me with a relaxing environment throughout the course of this work. I also want to thank the members of the Distributed Database Systems Research Laboratory for the useful discussions.

I would like to thank the other members of my examining committee, Dr. W. Armstrong, Dr. J. Schaeffer and Dr. P. Dorsey for their helpful comments.

I must thank Dr. W. Dobosiewicz and Dr. J. Schaeffer for helping me to solve some technical problems in the early stage of this research. Thanks are also due to Keith Fenske and Andy Nowacki for proofreading.

Contents

1 Introduction **1**

1.1 Introduction 1

1.2 Problems in Centralized DBMS 2

1.2.1 Buffer Pool Management 2

1.2.2 The File System 3

1.2.3 Process Control 4

1.2.4 Transaction Management 5

1.3 Problems in Distributed DBMS 6

1.3.1 Architecture 6

1.3.2 Interprocess Communication 7

1.3.3 Transparency and Naming 8

1.3.4 Fault Tolerance 10

1.4 Scope and Organization of the Thesis 11

2 Related Research **13**

2.1 Background 13

2.2 Hydra 15

2.2.1 Hydra Object 16

2.2.2 Naming and Protection 16

2.2.3 Communications 17

2.3 Eden 18

2.3.1 Eden Objects 18

2.3.2 Communications 18

2.3.3 Naming and Protection 19
2.3.4 Fault Tolerance 19
2.4 Amoeba 20
2.4.1 Communications 20
2.4.2 Protection and Naming 20
2.5 Clouds 21
2.5.1 The Clouds Paradigm – Object/Action Model 22
2.5.2 Communications 22
2.5.3 Protection 23
2.5.4 Fault Tolerance 23
2.6 Mach 24
2.6.1 Tasks and Threads 24
2.6.2 Communications 24
2.6.3 Matchmaker 26
2.7 UNIX 26
2.7.1 Protection 26
2.7.2 Communications 27
2.7.3 Distributed UNIX 28
2.8 V Kernel 29
2.8.1 Communications 29
2.8.2 Naming 30
2.9 Locus 30
2.9.1 Naming 31
2.9.2 Distributed File System 31
2.9.3 Communications 32
2.9.4 Fault Tolerance 33
2.10 Summary 33

3 FLEX Architecture **35**

3.1 FLEX Goals 35

3.2 FLEX Paradigm 36
3.2.1 Client-Server Approach 37
3.2.2 Separation of Policies and Mechanisms 38
3.2.3 The Object Model 39
3.2.4 Overall Architecture 40
3.3 Kernel Design 41
3.3.1 Object Manager 42
3.3.2 Inter-Object Communication Manager 43
3.3.3 Page Manager 44
3.3.4 Device Manager 44
3.3.5 Local Process Manager 45
3.4 Non-Kernel Design 45

4 Object Manager **47**
4.1 Object Model 48
4.1.1 Type Hierarchy 50
4.1.2 Active Versus Passive Objects 50
4.2 Object Management 52
4.2.1 Object Naming 52
4.2.2 Object Addressing 54
4.2.3 Object Descriptor 57
4.2.4 Object Manager operations 58

5 Access Control and Protection **60**
5.1 Protection of Capabilities 61
5.1.1 Background 61
5.1.2 FLEX Approach 62
5.1.3 Advantages of FLEX Approach 66
5.2 Protection of Objects 66
5.2.1 Domain Switching 68
5.2.2 Protection Enforcement Using Object Coordinator 70

5.3 Protection of Users 72
5.4 Capability Operations 73

6 Context Switching and Scheduling **74**
6.1 The Process Model 74
6.2 FLEX Design 77
6.3 Retrospective 79

7 Implementation **81**
7.1 Environment 81
7.2 Simulation Tactics 82

8 Conclusion **84**
8.1 Future Research 85

Bibliography **87**

List of Figures

1.1 Process Model versus Server Model 5

2.1 A Capability 14
2.2 An Access Matrix 15
2.3 Local Message Communication 25
2.4 Transparent Network Communication 25
2.5 Layers of the UNIX Operating System 27
2.6 Open Protocol 32

3.1 The Client-Server Model 37
3.2 FLEX Architecture 41

4.1 Example Object Structure 49
4.2 Object Hierarchy 50
4.3 Active Object Structure 51
4.4 Object Addressing 55
4.5 Object Directory 56
4.6 Structure of an Object Descriptor 57

5.1 FLEX Capability 62
5.2 User Capability 62
5.3 C-list in FLEX 63
5.4 Example showing object sharing with different capabilities 64
5.5 Structure of the Operation Table 70
5.6 Structure of the Access Table 71

6.1 Process State Diagram 75
6.2 Object Invocation 79

Chapter 1

Introduction

1.1 Introduction

In recent years, there has been considerable interest in developing distributed computing systems. As the cost of microprocessors declines, for a large class of applications, it becomes preferable to use small interconnected systems instead of centralized mainframes. The most commonly mentioned advantages of distributed systems are improved performance through the parallel application of computing resources, increased availability since the failure of one part of the system may not affect the operation of other parts, and incremental growth. These advantages, however, come at the expense of increased complexity of the system and the associated difficulty in managing it.

One of the applications of distributed systems is to support distributed databases. However, the kind of support required from the underlying distributed operating system (DOS) is a difficult issue. In fact, this problem already exists in a centralized setting. Conventional operating systems have been criticized for not providing adequate support for database management systems (DBMS) [Gra78,Sto81,TM82]. The major difficulty is the lack of a reasonable paradigm for co-operation between the OS and the DBMS. The result is a duplication of some common services leading to a degradation in system performance.

The FLEX project is an attempt to investigate the architectural as well as algo-

rithmic problems associated with providing adequate operating system support for distributed database systems. The design of FLEX and the prototype implementation of some parts are the concerns of this thesis. Neither the FLEX project, nor this thesis deals with the issue associated with determining the trade-offs between centralized and distributed computing; the distributed computing environment is a given for this research.

1.2 Problems in Centralized DBMS

As mentioned above, traditional database systems are built on top of existing operating systems that were not designed with the requirements of database systems in mind. Often the wrong service is provided and severe performance problems exist. The standard way of doing things in existing operating systems does not permit the DBMS to maintain a sufficiently fine grain of control. The result is that many of these services are re-implemented in the user space leading to a duplication of functions. In this section, we will summarize these problems. The discussion is based on Stonebraker [Sto81] as well as some recent refinements in [Ozs88].

1.2.1 Buffer Pool Management

Accessing data files from secondary storage is a very expensive operation. Therefore, most operating systems make use of a main memory buffer cache to improve the efficiency of accessing data files. The main memory buffer cache is divided into pages (called buffer pages). In dynamically allocating buffer pages, the OS assumes that page reference of processes are random and varying over time. However, it is possible for the DBMS to determine, with significant regularity, its reference to data pages which allows a more precise calculation of the buffer requirements [Cho85].

Along the same lines, it may not be useful for the OS to prefetch pages when it detects sequential access. Since the DBMS knows its reference pattern, it is in a better position to make that judgement. The reference string is known by the DBMS either because of query compilation (e.g. DB2 [Dat84]) or during execution (e.g.

INGRES [SKWH76]). Although the DBMS is in a better position to determine which page to prefetch, on a conventional operating system, there is no way to incorporate such knowledge.

Most conventional OSs implement virtual storage using a paging or segmentation system. Page replacement strategies like the least recently used (LRU) algorithm are generally used to decide which page should be removed from primary storage in order to make room for an incoming page. Such an algorithm works when processes exhibit a locality of reference in their page accesses. However in large databases, there is usually insufficient locality. For example, pages that contain information about the database organization itself should be kept in main memory. However, using a LRU algorithm, they will be replaced if some user data has to be brought in. Then immediately after, the database organization has to be read in again, thus causing continuous page faults.

Replacement algorithms that use delayed-writing create reliability problems in database transactions. Since pages might not be written to secondary storage immediately after a transaction updates, system malfunctions occurring at this moment will lead to an inconsistency in the database.

1.2.2 The File System

File systems on conventional operating systems like UNIX [1] [Tho78] have been criticized for limiting the performance of database systems. There are three major problems. The file system is made up of a sequence of logical blocks and the next logical block in a file need not be physically close to the previous one. A DBMS however, performs substantial sequential access. Given the fact that the OS provides no means of placing contiguous logical blocks together, a great number of disk arm movements are involved.

The size of logical blocks will affect the efficiency of data transfer. If the block size is small, we will need more I/O operations to transfer the data from the disk

[1] UNIX is a trademark of AT & T

to the main memory. The block size of the file system has been criticized for being too small for database applications. Nevertheless, this problem has been dealt with in some versions of UNIX [Bac86] by providing various block size: blocks can be 512, 1024 2048 or any convenient multiple of 512 bytes. Still there is no facility for supporting blocks that vary in size dynamically.

Since a file consists of a sequence of logical blocks, the file system must have some ways of keeping track of the blocks for each file. A common way is to use a tree structure [Tan87]. The problem associated with it is that DBMS always add another tree to support keyed access (e.g. B-trees). Obviously managing the same information with two different trees is a duplication of functions.

1.2.3 Process Control

In a single machine environment, there are two ways to organize a multi-user database: the process model versus the server model (Figure 1.1). In the process model, each database user runs in a separate process. These processes share the code segment of the database as well as some data structures. In the server model, there exists a single database process which acts as a server. This process handles requests from the users.

Three problems exist in the first approach. First, creating a process for each user request can be expensive, especially when the processes are big and contain a lot of state information (e.g. in IBM systems [Pet78]). Second, whenever a database process issues an I/O request, a context switch is performed. Again, the size of the process tends to make such switches inefficient. Third, since the database processes are just ordinary OS processes, they are scheduled by the OS scheduler. A problem arises if the OS deschedules a process that is holding a lock to a critical part of the database. Other database processes cannot execute because they cannot acquire the lock and very soon there will be a queue behind the locked resource. This is known as the *convoy phenomenon* [Bla79].

The server model can potentially solve these problems. This scheme does not need to create as many processes as there are users. As soon as the server finishes

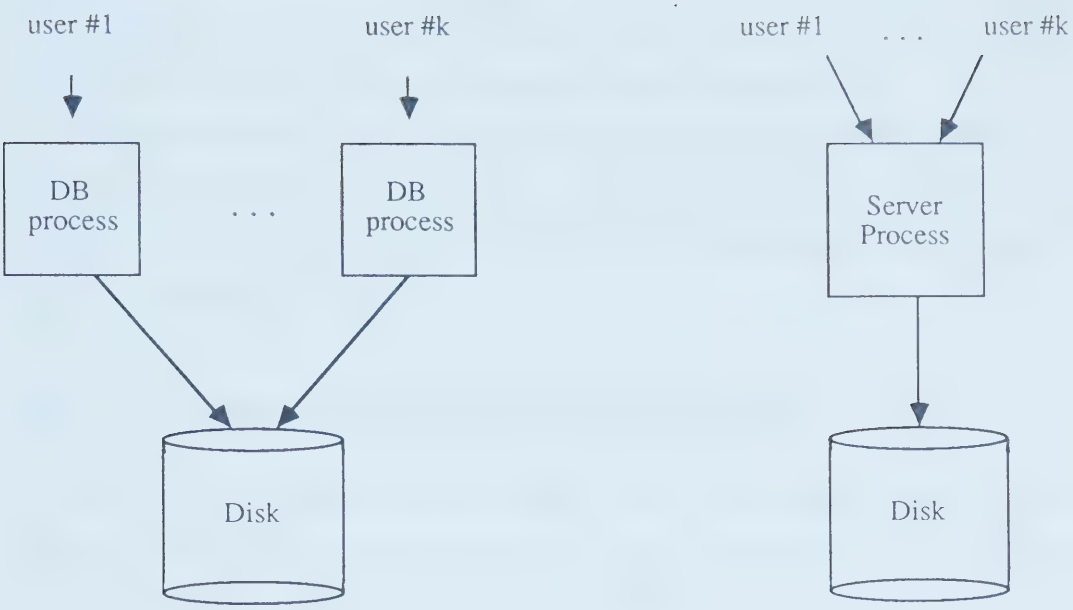

Figure 1.1: Process Model versus Server Model

servicing one user, it can service another one. Furthermore, the server can help avoid the convoy phenomenon by proper scheduling decisions. However, this model quickly runs into the problem of duplicating the OS functions because the server is doing its own multi-tasking and scheduling.

1.2.4 Transaction Management

A transaction is an atomic unit of execution, i.e. the sequence of operations involved is either performed entirely or not performed at all. This concept was developed to provide database reliability. It is related to the problem of recovery and concurrency control because atomicity must be preserved in case of failures and concurrent execution [CP84]. However, the mechanisms for concurrency control and recovery in conventional operating systems create problems for managing transactions.

For concurrent transactions that update shared data, facilities like locks must be provided to maintain data consistency. Most operating systems provide locking

for files. However only a few of them provide locking for pages or records. Clearly, if one wants to have more concurrent transactions, it is necessary to have smaller locks so that the shared data can be accessed by more transactions.

To provide atomicity, the transaction manager must be able to recover from crashes. Many DBMSs use intention lists or log files. These are implemented on top of the operating system because the OS recovery mechanisms do not support database transactions.

1.3 Problems in Distributed DBMS

The problems in the centralized DBMS also exist in the distributed DBMS. New problems arise because of the introduction of the communication network into the picture. The following sections discuss such problems.

1.3.1 Architecture

The traditional architectural models for operating system development cannot be applied in a distributed environment [Ozs88]. For example, a popular architectural model has been the layered approach. In a layered OS, each layer of software provides some service to the higher layers. However, there is considerable difficulty in applying this paradigm in a distributed environment because there are two physical resources: the computer facility and the network. The functionality required of the lowest layers of the operating system is quite different for the two resources. A separate set of protocols is needed to manage the network for which standards have been developed (e.g. ISO [Int83] for wide area network and IEEE 802 [Sta86] for local area network).

Attempts have been made to incorporate the network protocols into the layered operating system architecture [DB83]. The cost of such incorporation is a considerable increase in the number of layers of software that need to be implemented. Furthermore, user processes and new services such as D-DBM algorithms can only be implemented as further layers of software on top of the existing layers and so

there will be a performance cost. The issue of interfacing the database manager with such an OS architecture still has the problems discussed earlier. The design of the database algorithms will inevitably be limited by the design constraints of the OS.

1.3.2 Interprocess Communication

This is the mechanism that allows arbitrary processes to exchange data and synchronize execution. There are many forms of interprocess communication (IPC), among which are monitors, signals, files, shared segments, semaphores, etc. Many of these are not applicable in a distributed setting because the computers do not share primary memory. Two alternative approaches are *message passing* and *remote procedure calls*. Regardless of which one we use, several criteria must be satisfied in order to build any distributed application:

1. From the user's perspective, the access method for local or remote resources should be the same. That is, the syntax and semantics for issuing any requests should be the same no matter where the requests are issued. In this case, the network becomes transparent to the user. The issues of transparency will be discussed in the next section.

2. IPC should be efficient. Communication is probably the most important factor that affects the performance of distributed systems because it is the primitive for building distributed applications. As [TvR85] points out, there exist systems that require "the full computing power of its machines just to run the communication protocol".

3. IPC should be reliable. In particular, the communication should not lose messages and should be capable of returning all undeliverable messages to the sender. These are the minimal requirements for recoverability of D-DBMS's [SS81].

1.3.3 Transparency and Naming

A key concept in the design of a distributed system is transparency. At the extreme, the idea of transparency means that the use of multiple processors is invisible to a user. A user can approach a machine which is a node on a network and issue all requests as if the requests were located at that machine. This is easier said than done and many systems that do not fulfill this requirement have been built [TvR85]. In the following, we categorize the sorts of transparency that need to be provided in a distributed system.

In any operating system, there are many objects such as files, directories, segments that can be accessed by processes or users. To access such objects, the process or user must be able to specify them through some names. In a distributed system, it is important to be able to move files and processes from one machine to another to maintain a balanced load. To do so, the *location* of an object should be independent of the *name* of the object. As pointed out in [PW85], it is common to embed file names in programs. If a file name contains information about its location, then moving the file will make all previously correct software no longer operational.

Such name and location transparencies are not the only kinds that are desirable in a distributed system. To increase reliability and availability of services or data, resources are replicated across sites. Such replication should be transparent to a user so that he or she can access the resource as if there were only one copy. This characteristic is called *replication transparency*. The operating system should be responsible for maintaining consistency of the multiple copies and to choose the "best" one for any request.

To support D-DBMSs, two additional transparency requirements must be satisfied. The first one is *data independence* and the second is *fragmentation transparency*. In a centralized database, data independence is the fundamental transparency that is required. It refers to the sheltering of user applications from any change in the definition and/or organization of data. This requirement is naturally carried to the distributed setting. Fragmentation transparency becomes relevant in distributed database systems because it is common to divide the database relations

into smaller fragments. Such fragmentation should be invisible to the users who perform queries.

All the forementioned transparencies are required to support distributed database management. Even though all of these can be provided entirely by the D-DBMS, it is preferable that some of them be provided by the operating system. The data independence and fragmentation transparency are required by the database management system alone and should probably be provided within it. The rest should be provided by the operating system so that all the applications can make use of the facilities and customize them to their appropriate needs.

In order to provide the various types of transparency, services and objects must be named in a way so that remote access should not be any different from local access. A name should uniquely identify the entity it represents. In a centralized system, naming can be handled easily by using a name server. The name server will maintain a table or database that will provide the name-to-object mappings. Whenever a new name is created, it can be checked for uniqueness.

Such an approach can also be used in a distributed setting. Every service and object in the system will register with the name server. However, the location of the server becomes an issue. If the server is located on a particular machine, then it might easily become a bottleneck, especially when that machine becomes overloaded. Furthermore, any failure on that machine causes objects at other operational sites to be inaccessible because they cannot be named. An alternative way is to have multiple name servers, one for each machine. When a name is presented to one of the servers, it will find the corresponding object. If none can be found, the request will be broadcast to other name servers and a reply will be returned if there is a match.

The use of multiple name servers makes it harder to guarantee the uniqueness of names. For example, a user might register a file **tmp** with one name server and another user might register another file with the same name with another name server. To solve the problem, an appropriate naming scheme must be devised. One strategy is to use path names (e.g. machine1/user1/tmp). However, moving any

object will be difficult because the location information is embedded in the name. A second strategy is to have globally unique names, conceivably generated by the system. One problem with this approach is the guarantee of uniqueness. If a global generator is used, then one falls back into the problems of having a central name server. If not, then it becomes more complicated to guarantee that the name is unique.

1.3.4 Fault Tolerance

Distributed systems have the potential for a higher degree of *availability* and *reliability* than centralized systems. The reason is that the system can be made to tolerate both hardware and software errors by replicating resources. For example, in a centralized system, when a critical component, say the processor, goes down, the system will go down. This is not the case in a distributed system because there are multiple processors. Other parts of the system can continue to function and hence the availability of the system increases. Reliability is also increased by replicating information. If data is replicated across the sites, then the physical destruction of, say a disk, will not be disastrous. The replication of data is particularly useful in distributed databases because data retrieval can be performed on any copy. In short, a higher degree of fault tolerance is possible in a distributed system because resources can be replicated.

The replication of resources, however, causes new problems. Since data is replicated across machines, some means must exist to ensure that they are consistent. In particular, if one of the machines crashes when users are performing concurrent updates on the data, then an inconsistency will arise. The concept of a transaction can again be used, but here we are dealing with distributed transaction management. It has been observed [Gra78] that the structures needed for a process in a distributed transaction are almost identical to those needed for a centralized transaction. In particular, at each site, one can have a local transaction manager which implements local transactions. However, recovery mechanisms for distributed transactions will be more complicated because we have to deal with failures that occur at each site

as well as failures that occur in the communication between sites.

1.4 Scope and Organization of the Thesis

The FLEX project is an attempt to investigate the architectural as well as algorithmic problems associated with integrating distributed operating systems and distributed database management systems. The fundamental research issues in FLEX are (1) to investigate and experiment with a suitable architectural paradigm of cooperation between the distributed database system and the distributed operating system; and (2) to investigate and experiment with various management policies (e.g. transaction management, buffer management) in order to determine their suitability for distributed database functions; and (3) to identify, as a result of (1) and (2), the fundamental principles of OS and DBMS design that might facilitate better cooperation. The expected results of this research are two-fold: (1) the determination of the principles that are important in the design of operating systems for proper distributed database management systems support; and (2) the development of a prototype distributed operating system kernel that facilitates the development of distributed database management systems.

The research goals indicated above require the development of an operating system kernel from scratch. It is not possible to either investigate or solve the technical issues by simply attacking a component of the overall system. For example, it might be argued that the major issue relates to the design of a distributed file system and that the efforts should be concentrated on designing an efficient distributed file system that supports database access methods. However, that does not solve the architectural issues related to integrating distributed operating systems and distributed database management systems.

A number of graduate students are currently involved in the FLEX project, each working on different parts of the system. The parts that this thesis deals with are Object Management, Protection and Scheduling. The thesis is organized as follows: Chapter 2 reviews some distributed systems that have been constructed as research

projects. Chapter 3 discusses the architecture of FLEX and the functionalities of various components. Chapter 4 presents the design and implementation of the object manager. Chapter 5 discusses the access control and protection mechanisms. Chapter 6 discusses scheduling and dispatching. Chapter 7 presents the prototype implementation. Finally Chapter 8 summarizes the thesis findings and provides a discussion of the ongoing research.

Chapter 2

Related Research

2.1 Background

This chapter is a survey of some of the distributed systems that have been built. A distributed system usually refers to a system that runs on a number of processors connected by a network. The processors are loosely-coupled which means that they normally do not share memory. Furthermore, the users cannot tell which machines they are using. Based on these definitions, some of the systems below should not be included. We choose to include them because of their particular features and to make the discussion complete.

The systems can be classified into two categories: object-based vs non-object-based. Object-based systems provide a higher-level of abstraction than non-object-based systems. For example, in object-based systems, it is common to provide a basic set of primitive objects so that higher-level object types can be constructed using combinations of them. The two systems also differ in *naming* and *protecting resources*. For example, using the object model, each resource in the OS (e.g. processes, files, I/O devices etc.) is treated as an object and it becomes appropriate to name objects in a uniform manner. In the other model, it is common to have different name interpretations for different resources. For example, processes can be named by integers; files can be named by ASCII strings that are meaningful only to the file system.

Capabilities [DH66,Fab74] are often used in conjunction with the object-model because of its ability to provide uniform naming. A capability can be thought of as a token or key that gives the bearer permission to access an entity in a computer system. It can be implemented as a data structure that contains two fields: a unique object identifier and access rights as shown in Figure 2.1.

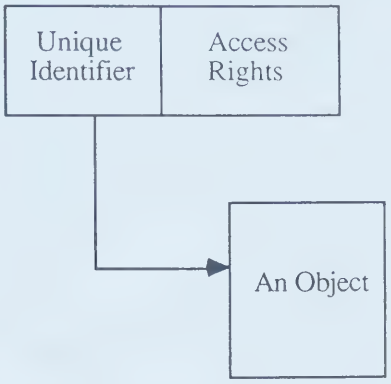

Figure 2.1: A Capability

Since capability will be frequently encountered in the subsequent discussions, we will elaborate on its usage. Aside from addressing objects, capabilities form the basis for object protection. The access rights part of a capability defines the operations that can be performed on the object addressed by the capability. For example, the access rights can permit read access to a file or write access to a memory segment. Each user or program in a capability system has a *list of capabilities*. These capabilities define the set of objects that the user or program is permitted to access.

Using capabilities is only one way for protecting information. Another alternative is to use *access control lists*. Figure 2.2 depicts parts of an access matrix which shows the privileges for each user. Using this approach, an access list is kept for each object in the system. For example, access list for File1 is John(R), Mary(RW), where R indicates reading privileges and W indicates writing privileges. Each time a user attempts to access an object, the OS checks the access list associated with the object to see if the operation is legal.

		System Objects					
		File1	File2	Mailbox	Printer	a.out	
User Objects	John	R	R	send	print	execute	
	Mary	RW		receive	print	execute	
	Joe		R	send receive	print	execute	

Figure 2.2: An Access Matrix

To contrast the access control list approach with capabilities, we can slice up the matrix by rows. Here a *capability list* is associated with each user in the system. For example, the capability list for Mary is File1(RW), Mailbox(receive), Printer(print). To access an object, a user supplies a capability, which he or she has to acquire beforehand.

For object-based systems, we will study Hydra, Eden, Amoeba, Clouds and Mach. For non-object-based systems, we will examine UNIX, the V-kernel and Locus. Where possible, the following topics will be discussed: communications, naming, protection and fault-tolerance. In addition, important features of each system will be highlighted.

2.2 Hydra

Hydra [WLH81,WCC*74] is one of the first major capability-based operating systems. It was built at Carnegie-Mellon University as the operating system kernel for the C.mmp computer system. The C.mmp is a multiprocessor and consists of 16 PDP-11 minicomputers connected to 32 Megabytes of shared memory. Hydra

was designed to allow operating system experimentation. To do so, the designers implemented a kernel so that other operating system facilities can be constructed as normal user programs. *Mechanisms* are separated from *policies* [LCC*75] – the kernel provides the mechanisms for implementing various policies (e.g. low-level dispatching), whereas the policies (e.g. scheduling decisions) are implemented as user-level software.

2.2.1 Hydra Object

A Hydra object has three parts: a unique name which is a 64-bit value obtained from the system clock; a type that defines the nature of the resource represented by the object (e.g. file, page, process etc.); and a representation that contains the actual information of the object. The representation is divided into 2 parts: a data-part and a C-list. The data-part is a block of storage that can be read or written. The C-list contains capabilities for other objects.

2.2.2 Naming and Protection

Hydra goes into a great depth in designing its protection mechanisms [CJ75]. In accordance with the Policy/Mechanism separation principle, the kernel supports a set of flexible protection mechanisms so that a wide range of security policies can be provided to higher-level software (e.g. passwords, files). There are three essential elements for the protection mechanisms. The first is that the unit of protection is the object since this is the abstracted notion of an arbitrary resource. The second is that access to objects is controlled by capabilities. Finally each program should execute with the smallest set of access rights possible and each procedure should execute in a completely new run-time environment.

Hydra's capabilities contain an object's name and access rights. The access rights are made up of a generic rights field that can be applied to any object and an auxiliary rights field which is type-specific. Capabilities can only be manipulated by the kernel and hence they are protected from any forgeries.

Hydra's protection is enforced at procedure invocation time. The kernel provides 2 functions: CALL and RETURN which respectively allow a procedure to invoke a new procedure and to return to the caller's procedure. The CALL operation creates a new execution domain Local Name Space (called an LNS-object) and performs parameter passing/checking. A LNS is created for each invocation and disappears when the procedure terminates. It defines all the capabilities available to a procedure during the execution resulting from a particular invocation. The parameter checking verifies that the parameter capabilities have the expected type and required rights. Under certain circumstances, it performs rights amplification (an issue that we shall discuss later) so that the parameter capability will have greater rights. This facility allows the called procedure to perform operations on an object that are not permitted in the caller's procedure.

2.2.3 Communications

Hydra uses a message system for communication. There are four concepts involved:

1. *Port* - messages are sent between objects of type port, not between processes. A port provides the abstraction of a service and it can be shared.

2. *Channel and connection* - messages leave ports on output channels and arrive at ports on input channels. A "connection" between output channels and input channels can be defined. After a connection is established, we only need to specify the output channel, not the destination of a message.

3. *Message* - a message is a buffer for transmitting data, not the actual data.

4. *Reply* - one field in the message buffer is a stack of reply frames. Whenever a message is sent, a new reply frame is pushed on the stack and contains information about the senders. A reply operation pops off the top frame and uses its contents to determine the return destination. Hence it is possible for a message to be forwarded many times and a reply returned to the sender.

2.3 Eden

The Eden System [LLA*81,ABLN85] was developed at the University of Washington in Seattle to build an "integrated distributed" computing system. The idea is based on the observation that contemporary systems are either centralized, where there is a high degree of integration but poor support for personal computing, or distributed where there is good support for personal computing but a low degree of integration. It uses the object-oriented approach and a distributed application is built as a collection of Eden objects called Ejects. Eden is implemented on top of UNIX - the Eden kernel on each machine is written in C and runs as a UNIX process. Applications are written in the Eden Programming Language (EPL) which is based on Concurrent Euclid [Hol83]. Each Eject executes within a separate UNIX process with its own address space. The Eden kernel manages this process using UNIX facilities.

2.3.1 Eden Objects

An Eden object has four parts: the name, which is a unique binary identifier for the object; the representation, which consists of data and capability segments, forming the object's long-term state; the type, which describes the routines that maintain the abstraction of the object or contains a capability for a type manager that defines operations allowable on the object instance; and a short-term state, which includes temporal data, the processor state, etc.

2.3.2 Communications

Active Ejects communicate with each other by sending messages. Each Eject consists of a number of processes. When a process in one Eject invokes another Eject, that process blocks until the invocation completes, but other processes within the invoking Eject can continue to run.

The invoked Eject normally has a process within it that has blocked itself waiting for a message to arrive. When a message arrives, a special dispatcher process will

check to see which processes are blocked waiting for the message. If one is found, it is run and a reply is sent back to unblock the caller. If a process is not found, the message will be queued.

2.3.3 Naming and Protection

In Eden, a message can be sent without knowing the target object's location. To invoke an Eject, a process must have a capability for the target Eject. The capability contains both the unique identifier of the Eject and a set of access rights.

Each Eject keeps capabilities in its own address space. The kernel maintains a protected table containing copies of these capabilities. Whenever a capability is used (e.g. in an invocation), it is checked for existence in the table. However, if an Eject wants to examine the fields of a capability (e.g. its access rights), communication with the kernel can be avoided. The operations for such examinations are kept in a library which is linked with each Eject. Therefore an Eject can examine the capabilities it kept by using a procedure call instead of a message to the kernel.

2.3.4 Fault Tolerance

Ejects can be either active or passive. When an Eject is created, only the active form exists. Multiple processes exist within the Eject but the Eject does not exist on permanent storage; the Eject would vanish if any crashes occurred.

An active Eject, however, can execute a checkpoint sequence which creates a passive representation. The passive representation is a data structure that contains information to restore an Eject after the system crashes. Thus an Eject can achieve a high degree of reliability by checkpointing itself periodically. However such operation is expensive especially for objects that make small changes all the time because the checkpointing requires writing the entire object to the disk.

The checkpointing action is atomic but the Eden kernel does not support atomic actions. Any applications that require atomicity must build them out of Ejects.

2.4 Amoeba

Amoeba [TM82,MT86] is developed at the Vrije Universiteit in Amsterdam with the goal of investigating capability-based object-oriented system issues. A lot of research effort has been concentrated on providing a uniform naming and protection mechanism. Attempts are made to keep the kernel small and to run as much of the OS as possible as user processes. Currently the system runs on a collection of 24 Motorola 68010 computers connected by a 10-megabyte-per-second local network.

2.4.1 Communications

The basic paradigm in Amoeba for modeling interprocess communication is the service. A service is defined by a set of commands and responses. The service is implemented by one or more server processes. Each server listens to certain ports for incoming requests. When a client process A has work for a service S, A sends a message to one of S's ports. Every port has 2 names, one for clients to send to and one for servers to read from. Port names are long bit strings, chosen randomly from a sparse address space.

After making a request, a client blocks until it receives a reply. To make communication simple, the system does not use any "connections" or virtual circuits (i.e. every request-reply pair is self-contained); processes will not go into the trouble of setting up, using and destroying a connection. Although the basic message primitives are blocking, emergency messages can force the termination of services.

2.4.2 Protection and Naming

In Amoeba, each object is owned by some service and is managed by the corresponding server process. There are 2 levels of protection in Amoeba:

- ports for protecting access to servers.
- capabilities for protecting access to individual objects

The protection of ports is based on port numbers - a client cannot send messages to a port without knowing its number. To prevent users from forging port numbers, they are chosen from a sparse address space. If the port number is N bits, with N sufficiently large, and only a small fraction of the 2^N possible ports are actually used, then the chance of forgery is small.

Obviously a server is not required to carry out a request for a client just because he or she knows the port number. For example, a file server will not write to a file object if the client lacks the appropriate permission for the file. Therefore, capabilities are used to protect individual objects. When an object is created, a capability is returned to the creator. An Amoeba capability has four fields:

1. server - the port number for clients to send to
2. object - an internal number that the service uses to identify the object
3. rights - what operations may be requested by this capability
4. check - a large random number for authenticating the capability.

A capability gives the bearer permission to perform operations on the object. Capabilities are kept in user-space and are protected by cryptography, which is why a *check field* is used. Each server, however, can choose its own protection scheme. More details can be found in [TMvR86,MT84].

2.5 Clouds

The Clouds Project [DLS85] is initiated at Georgia Tech. Its goal is to use state-of-the-art techniques to design a fault-tolerant distributed operating system kernel that can run on conventional, unreliable hardware. The fundamental building blocks in Clouds are objects, actions and processes. The current prototype runs on three Vax/750 computers each with 3 Megabytes of memory and connected by an Ethernet.

2.5.1 The Clouds Paradigm – Object/Action Model

A Clouds object contains both data and code. Each object has its own virtual address space and is passive; i.e. no active processes reside permanently inside an object. However an object has defined entry points and these correspond to the start of the code that can be invoked by processes. Concurrent processes can run within an object by invoking the object at different entry points. Clouds objects are permanent - an object exists until explicitly deleted and all modifications to the data in an object are retained.

Actions consist of *invocations of operations* defined in objects. Object invocation is the basic activity in the system. An action transforms the state of an object from one consistent state to another. Therefore it is defined to be atomic - either all the invocations occur and run to completion or nothing occurs. Actions need not only perform sequential invocation, they can fork several processes to perform concurrent invocation. An action terminates only if all of its components are successful. Otherwise, all the updates are rolled back and hence consistency is maintained in objects.

Since rolling back may be costly in long actions, Clouds provides subactions which are created by actions. A subaction executes independently, it may commit or abort and the termination status is returned to its parent. Updates performed by committed subactions become permanent only if the top level action completes successfully.

2.5.2 Communications

As mentioned, object invocation is the basic activity in the system. To make it efficient, it is implemented as a low-level primitive. Two systems, the Storage Management System and the Object Management System provide the necessary support [PD87]. The Storage Management System is responsible for locating an object at a particular site. It uses three data structures: the Active Object Table (AOT), the maybe table, and the partition directories. If an object is found in

the AOT, then the search succeeds. Otherwise, the maybe table is searched. The maybe table provides efficient membership testing and indicates whether the object may or may not be at the current site. If the object may not be local, the disk partition directories are consulted and such a query is more expensive.

The Object Management System is responsible for handling object invocations. It will format the parameters, validate the capabilities (see Sec 2.5.3) and operation, and call on the storage manager to determine whether the call is local or remote. If the object is local, invocation can proceed. Otherwise, a remote procedure call is necessary. In either case, the object descriptor for the object is added to the local AOT.

2.5.3 Protection

Object invocation in Clouds is capability-based. A capability contains the name, some book-keeping information and the access rights allowed on an object. The protection scheme in Clouds is to match object capability with action capability when the action invokes the object. Invocation is allowed only if a match occurs. In this scheme, an action that has valid access to an object can give such privilege to other actions. However the recipient can only identify the object and does not have access privilege to the object. Access privilege is only granted by the kernel by capability matching. It is not clear how the kernel distinguishes the access rights of a valid capability with the access rights of a traded capability.

2.5.4 Fault Tolerance

In Clouds, both hardware and software malfunctions are handled by the action paradigm. The net result of any malfunction is the abnormal termination of actions. All actions running on the problem machine are automatically aborted. In addition, other actions running at other sites on behalf of the problem machine are also aborted. A monitoring scheme is implemented to track down the actions at other sites.

2.6 Mach

Mach [JR86,Ras86] is a multiprocessor operating system currently under development at Carnegie Mellon University. It currently runs on the four-processor VAX 11/784 with 8 MB of shared memory. In addition, it runs on most VAX architectures. Although Mach is not a distributed operating system, we choose to include it because the basis of Mach's kernel is to provide abstractions making it possible to run varying system configurations on different classes of machines, which include large general purpose multiprocessors, networks of smaller processors and individual workstations. The Mach's kernel supports four basic abstractions: task, thread, port and message. Task and thread define the execution environment, port and message define the interprocess communication facility. Mach provides a uniform reference to object using capabilities.

2.6.1 Tasks and Threads

It has been observed that UNIX processes carry a high overhead; resources like the page tables as well as process slots are tied up for each fork operation. To overcome this, Mach divides the process abstraction into two: task and thread. A task is a high-overhead object, much like a process, but a thread is a relatively low-overhead object. Multiple threads can execute within a single task. Parallelism is achieved by running the multiple threads within the same task simultaneously on the multiprocessors.

2.6.2 Communications

A port is a protected kernel object into which messages may be placed and later removed. Ports may have any number of senders but only one receiver. Access to a port is granted by receiving a message that contains a port capability.

Messages are used for communication between threads. A message is a variable size collection of typed data objects. It may be sent synchronously or asynchronously.

The majority of message communication occurs between objects on the same machine (Figure 2.3). The Mach kernel, by itself, does not provide any mechanisms for supporting communication between machines. However, using port and message abstractions, it is possible to build a transparent network communication using a message server (Figure 2.4). There is a message server on each machine and it provides transparent transmission of messages to remote ports.

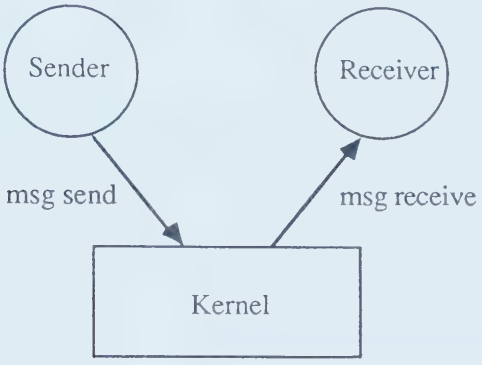

Figure 2.3: Local Message Communication

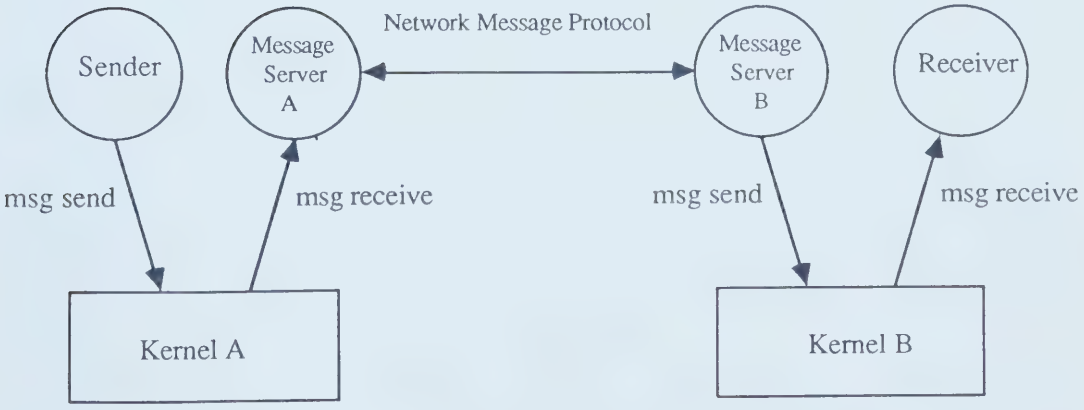

Figure 2.4: Transparent Network Communication

2.6.3 Matchmaker

To provide support for multiple programming languages, Matchmaker is developed as an interface specification language. It compiles interface definitions into remote procedure call stubs for various programming languages like C, Common Lisp, Pascal, etc. The Matchmaker code provides run-time support for type-checking, type-conversions, communications and exception-handling.

2.7 UNIX

The first version of UNIX was developed at Bell Laboratories in 1969. The UNIX OS is configured as a layered OS and the layers are depicted in Figure 2.5. Everything below the system call interface and above the hardware interface is the kernel [Tho78]. The kernel provides all the OS features including scheduling, interprocess communication, file management, paging, swapping, device drivers etc. The details of these can be found in [QSP85,Bac86].

Many versions of UNIX have been developed, among which the popular ones are System V and Berkeley UNIX. Although neither of them are DOS by themselves, there are recent attempts to build distributed facilities on top of them. We base our discussion on Berkeley UNIX because of its internetwork facilities.

2.7.1 Protection

The UNIX system recognises every user by a user-id (uid). Furthermore, each user is assigned a group identification (gid) that places him or her in a class of users. The file system, the execution of processes and UNIX in general, determine the protection domain by means of the uid and the gid.

The system uses the uid and gid of a process to determine the list of resources that it can access while it is executing. The resources may include files, directories, I/O devices, memory segments, etc. In addition, whenever a process issues a system call, it switches from user mode to kernel mode. In kernel mode, the process can

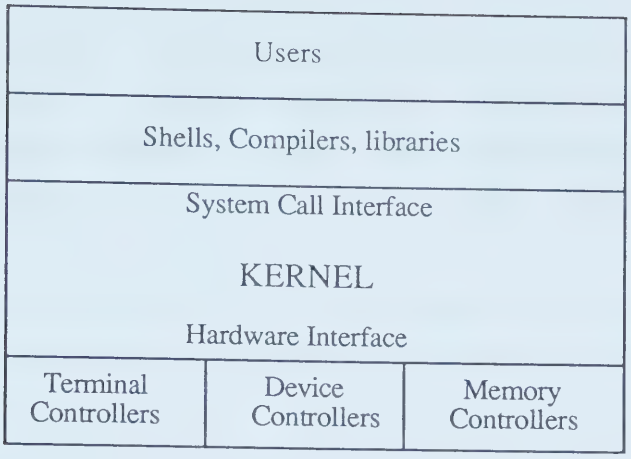

Figure 2.5: Layers of the UNIX Operating System

access those kernel structures that are inaccessible when it is executing in user-mode.

File access rights in UNIX are divided into read, write and execute. The access rights can be set independently for three classes of users. The classes are: owner, who owns the file; group, those who belong to the same group as the owner; and others, those that the world belongs to. An access list is kept for each file, and every time a reference is made, the OS checks that the reference has the necessary permissions.

2.7.2 Communications

Standard UNIX supports pipes and signals for communication. A pipe permits a stream of data to be passed from one process to another. The problem with pipes is that only related processes can communicate via them, i.e. only the descendants of a process that issued the pipe call can share access to the pipe. Arbitrary processes can communicate by sending signals using the **kill** system call. Although there are a number of different signals, none of them carry any parameters. Therefore no messages can be delivered to a process using signals. A process can only use signals to inform other processes of some events. Thus a signal is very much like an interrupt.

Berkeley UNIX provides a better form of communication known as sockets which can be extended across a network. A socket is an end point of communication. Two or more sockets must be connected before they can be used to transfer data. Connections can occur on the same machine as well as across machines. The latter type uses protocols like the Transmission Control Protocol or Internet User Datagram Protocol.

There are several types of sockets, each representing different services. For example, SOCK_STREAM socket provides byte-by-byte stream communication. SOCK_DGRM provides datagram transmission and does not guarantee delivery. SOCK_RAW gives users high degree of control over message transmission.

2.7.3 Distributed UNIX

There are several attempts to enhance Unix with distributed facilities, the most notable one being Locus, which is a true DOS. Locus is discussed in Sec. 2.9. Other attempts include providing facilities to mount a remote file system, distributed/extended file system, etc. We look at one of the recent approaches: the Sun Microsystems' Network File system (NFS) [SGK*85].

The NFS was designed to make sharing file systems easy in a heterogeneous machine environment. It provides a high degree of file access transparencies. A remote file system (e.g. VMS filesystem) can be mounted on a local UNIX system in the same way as a file system on the local disk can be mounted. To the user, the file on the remote file system is just the same as a file on the local file system.

To implement the NFS, a new interface was added which consists of two parts: the Virtual File System (VFS) interface defines the operations that can be done on a file system; the vnode interface defines the operations that can be done on a file within the file system. There is one VFS for each mounted file system and one vnode for each file or directory. Using this abstraction, the kernel can install new file systems in much the same way as new device drivers are installed.

2.8 V Kernel

The V kernel was developed at Stanford University [Che84]. It was motivated by the economies and facilities of personal workstations as well as local networks. The goal is to build a kernel that provides good system and good kernel performance. V is implemented on a collection of Sun workstations, interconnected by a 3 or 10 Mbyte Ethernet. Each station runs an identical copy of the V kernel. The kernel has three major components: interprocess communication, a kernel server which provides a message interface to the process control and memory management services, and the device server which provides access and management to I/O devices.

2.8.1 Communications

The V kernel implements message-based communication. A client process uses a **send** operation which transmits a message to a server process. The client process then blocks. The message eventually causes the server to execute a **receive** operation to receive the message. Finally, the server executes a **reply** to send a reply message back to the client and unblocks it.

There are two parameters for the **send** operation: the first one is a 32 byte message. A reply from the server will overwrite this message. The second parameter is a 32-bit process identifier. It is globally unique and contains location information: the high-order 16 bits form a logical host identifier and the low-order 16 bits form the process identifier.

In an environment where there are many cooperating processes, it is often useful to be able to view some set of processes as a single logical entity. The V kernel supports the concept of process groups [CZ85]. A V process group is a set of one or more processes that may be distributed on different machines. A process group is identified by a group identifier. Sending a message to a group is similar to sending it to a process, except that a group identifier is used instead of a process identifier. The sender blocks until at least one process from the group receives the message and sends back a reply.

The obvious advantage of process groups is that we can have well-known group identifiers. For applications that implement operations using multiple processes (e.g. database applications), it is easier to invoke the whole application using a group identifier than having to remember each individual process identifier in order to use it. In addition, it is shown that applications that perform real-time updates of replicated data run more efficiently using group operations.

2.8.2 Naming

The V kernel has three levels of naming [CM84]. At the lowest level is the IPC Naming: the 32-bit process identifier (pid) uniquely identifies a process. At the next level is the Service Naming: server processes that implement a particular service (e.g. printing, clock) can register a symbolic name with the kernel so that the client can access the service using the name instead of the pid. The advantage of this mechanism is that clients are sheltered from changes in process identifiers. For example, if the processes that implement the device drivers crash, then a new set of processes will reboot the drivers. The process identifiers will be different but the clients need not be aware of it.

The top level naming is Object Naming. There are two types of objects: temporary and permanent. Temporary objects exist only during the execution of programs. They are named by a 16-bit numeric identifier that is generated by the server that creates the object. Permanent objects exist beyond the program execution and are named by either users or applications using high-level character string names whose interpretation depends on the context in which the name is used.

2.9 Locus

The Locus system [PW85] was developed in UCLA. It started in the late 1970's as an attempt to understand how application software and computer networks should interact. Locus is a distributed version of UNIX. As a result, it is compatible with both Berkeley UNIX and System V. Central to the Locus architecture is its

distributed file system. It is fully transparent and highly reliable. In addition, Locus extends System V interprocess communication to increase transparency [Fle86].

2.9.1 Naming

In order to discuss file access, we have to understand the Locus naming mechanism. The name space in Locus is organised around the concept of a filegroup. Each filegroup is divided into two parts: a small set of file descriptors and a large number of data blocks. A file descriptor contains pointers to the data blocks which compose the file. A file name is specified by a <filegroup number, file descriptor number> pair. A filegroup can be replicated. Files may be replicated only at sites that store the containing filegroup.

2.9.2 Distributed File System

The Locus file system has three main characteristics. First, a single tree structure is used to cover all objects in the file system on all machines. That is, there is only one logical root in the whole network. Thus each entity or resource in Locus can be referred to by its pathname, much like that of a centralized UNIX environment. Second, Locus supports file replication and the system maintains consistency among the copies. Third, Locus provides a synchronization policy that is upward compatible with the total lack of synchronization in standard UNIX. Since Locus introduces data replication, synchronization becomes a necessity. Locking can be done at both the record and the file level.

A file access can invoke three logical sites:

1. using site (US): the site that issues the request to open a file.

2. storage site (SS): the site at which a copy of the requested file is stored and has been selected.

3. current synchronization site (CSS): for a given file group, one site is designated as the current synchronization site (CSS) and is responsible for coordinating

access to files contained in the associated filegroup.

The protocol for opening a file is depicted in Figure 2.6. Briefly, the open call is issued at the using site with a file name as a parameter. The CSS is determined, if it is local, a procedure call is made. Otherwise, a message is sent to the CSS. The CSS selects a storage site and sends it a message. If the storage site decides to provide the service, it replies to the CSS, which notifies the using site. The open call is then completed and processing at the using site can proceed as if the file had been local.

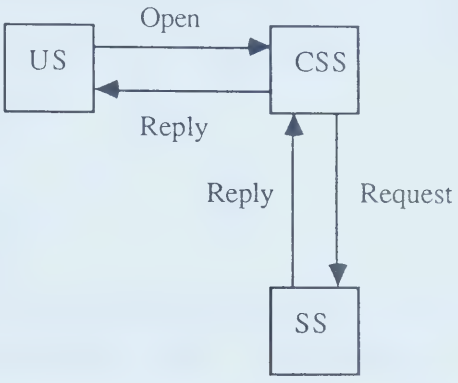

Figure 2.6: Open Protocol

2.9.3 Communications

The Locus IPC was divided into two parts:

1. the naming components that are used to maintain the IPC name space, and
2. the functional components that implement the message, semaphore and shared memory subsystems.

This separation allows Locus to implement the naming components outside of the kernel - the database of IPC names is stored in one special server called the IPC name server (IPC_NS). The reason is based on the observation that locating a name is less frequent than using the functional operations since names are typically

used to refer to objects that will be repeatedly used. Furthermore, performance can be optimized for the functional components because they are in the kernel itself.

2.9.4 Fault Tolerance

Reliable operations of the IPC_NS is essential for IPC services. To guarantee this, when the current IPC_NS fails, Locus will select another site to support a new IPC_NS. The newly selected site has an empty database and information has to be reinserted into it. To provide this facility, the kernel of each site maintains the message queues stored at that site. By polling all sites, it is possible to reconstruct the contents of the IPC_NS. The polling can be done at the user level so that no special kernel code is needed.

2.10 Summary

The eight systems that we discussed in this chapter share many common features but also differ in some significant ways. Communication is the most important aspect of building a distributed system and this is reflected by the the research effort that has been devoted to it in all systems. All the systems attempt to provide transparent communication. Efficiency is also emphasized by implementing it as a low-level primitive. The development of an interface language as in Mach is undoubtedly valuable because, without it, code has to be written in a single programming language.

In most of the systems, there are attempts to pull the traditional OS facilities out of the kernel. For example, in Hydra, policies and mechanisms are separated; in Clouds, Eden and Mach, primitive building blocks or abstractions are defined so that higher-level objects can be built; and in Locus, the name server is implemented in user space.

A different naming scheme is employed by each system. Within the object-based domain, capabilities are used to name objects. However they differ considerably in their formats as well as in the way they are protected. In the non-object-based

domain, various name formats might exist, e.g. process identifiers, symbolic names, pathnames and some forms of mapping are provided by the system.

Chapter 3

FLEX Architecture

3.1 FLEX Goals

The fundamental goal of FLEX is to provide adequate operating system support for distributed database systems within a general purpose distributed computing environment. The insistence on a general purpose computing environment should be emphasized. It is our belief that the most prevalent computing environment for DBMS applications will continue to be general purpose operating systems running on general purpose computers. Although database machine research has been going on for almost a decade, the research has not been translated into a significant commercial presence and the current market for such machines is quite small. On the contrary, it has been suggested that the database machines of the future will function more as file servers on a network instead of as stand-alone computers [Tay87]. Therefore, it is essential to investigate the limits of what can be accomplished within such an environment. Furthermore, it is important to find a suitable paradigm of cooperation between the distributed database system and the distributed operating system because of the problems described in Chapter 1.

FLEX attempts to keep the operating system kernel to a reasonable size. Having a small kernel enables the kernel to be memory-resident on a variety of computers. Most of the distributed computing environments are heterogeneous: there might be mainframes as well as workstations connected to the network. Some of the

workstations might have limited memory. By keeping the kernel small, the kernel can reside in main memory on all machines and thus achieve better performance.

Another goal of FLEX is to investigate an architecture that is easily extensible. An extensible architecture makes it easy for users to add new system software or hardware to the system. In conventional operating systems, where all components like scheduling, file system, memory management, device drivers etc. reside inside the kernel, it is impossible to change any of these without stopping the entire system. Furthermore, after the changes are made, a recompilation or relinking is needed before the system can be restarted. FLEX attempts to overcome these problems so that operating system modules can be installed or changed while the system is running.

The system will provide fully transparent access to system entities as well as reliable operations. However, FLEX does not attempt to be compatible with any existing operating system. Since this is a research project for investigating the best way to interface the operating system with the database systems, any attempt at compatibility will be too restrictive.

FLEX is designed to work in a broadcasting local area network environment. Therefore conformity with the IEEE 802 Local Area Network standard at the Logical Link Control Layer for remote communication [Ins85] is maintained.

Finally, we expect FLEX to serve as a testbed for experimentation regarding a number of issues associated with distributed database management and distributed computing. For example, management policies for transactions, buffers and scheduling etc. will be investigated.

3.2 FLEX Paradigm

FLEX architectural paradigm can be summarized as follows: it uses the client-server approach for co-operation between the DBMS, the OS and the computer network; it follows the separation of policies from mechanisms principle; and it uses the object model as the overall implementation technique. The details of each of these are

discussed in the following sections.

3.2.1 Client-Server Approach

In the client-server approach, the operating system is organised as a group of servers, each of which implements a specific service. Thus the kernel is another server and it interacts directly with the hardware. To request a service, a client sends the request to a server, which then does the work and sends back a reply. Figure 3.1 depicts this model. Note that the communication between the client and the server goes through the kernel. The reason will become obvious in subsequent paragraphs.

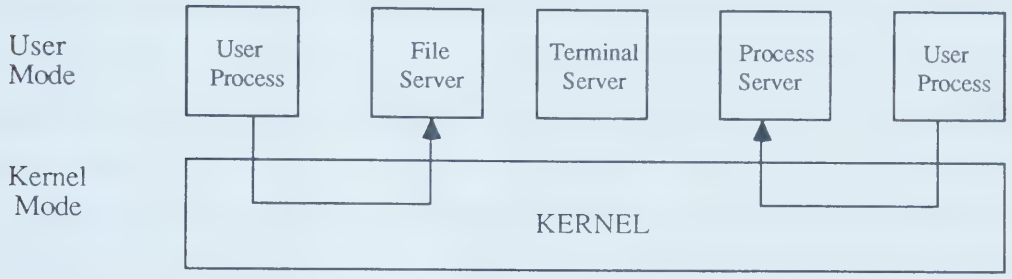

Figure 3.1: The Client-Server Model

By splitting the traditional kernel into various parts, each running as a server, the kernel becomes fairly small. Each part is also small and becomes more manageable. Since all the servers are running as user-level processes, it is possible to add/modify/delete any of them without affecting the operation of the whole system. Thus software can be developed incrementally and installation will be transparent to the user. However, because a server might not be operational while the system is running, communication between servers as well as communication between clients and servers cannot be direct. The communication has to go through a third party which maintains the address of the existing servers in the system. In Figure 3.1, such communication is trapped by the kernel. Alternatively one might have another user-level server that traps the communication.

This approach also facilitates the use of a dynamic layering approach to operat-

ing system design instead of the traditional static layering approach. If a software module requests the service of another module, which, in turn, can request the services of some other modules, a layering can be established. However, such layering occurs in a dynamic way and is specific to the requirements of that software. Other software does not necessarily have to conform to this particular layering structure. For example, if the DBMS does not need the services of a file system, then the file system does not exist as part of the operating system as far as the DBMS is concerned.

This approach is particularly attractive in a distributed system because the servers can be distributed to various machines to maintain a balanced load. Selective replication of some service is also possible. With a transparent IPC mechanism, a client process does not need to know the location of a server process. To the client, local and remote services are handled in the same way: a request is sent and a reply comes back.

Although this approach is appealing, difficulties arise when we try to run all OS features as user-level processes. For example, low-level context-switching which deals directly with machine registers is almost impossible to run in user-mode. As we have seen, the kernel has to keep track of the location of the servers. Communication becomes more complicated because we no longer have a big kernel where all the modules are linked together and procedure calls can be made. The challenge then is to build a minimal kernel that provides the necessary and sufficient support. To make this possible, we have adopted the separation of policies and mechanisms principle.

3.2.2 Separation of Policies and Mechanisms

The principle of separating policies from mechanisms was first argued for in Hydra [LCC*75] and has now been adopted by others [Tan87] as a way of building a minimal kernel. Policies that govern the allocation of resources are defined at the user level. Mechanisms for implementing these policies are provided inside the kernel.

Such a separation allows the system developers to tune the policies to suit individual applications. In addition, multiple policies can exist for the same class of resources. This can solve some of the problems that we discussed in Chapter 1. For example, we can have a LRU page replacement policy for non-database applications and another policy for the database management system.

Although this seems to be a clean concept, putting it into practice is not trivial. There are two problems: first, given a particular piece of software, it is necessary to distinguish between those parts that contribute policies from those parts that contribute mechanisms. Such distinctions are of course relative and it is not surprising that, in some cases, the distinction might not be clear or might not exist. Second, there might be a considerable performance cost. The following example will illustrate these points.

In most OSs, the information about a process is usually kept in a process table which resides in the kernel. Consider the case where we move the memory management policies outside the kernel. Whenever a new process is created or an old process terminates, the memory manager has to update the process table. The result is a kernel call or kernel message which isn't necessary if the memory manager is located inside the kernel. Therefore, care must be taken to enforce the principle. This includes designing the data structures and algorithms to minimize communication with the kernel. For example, to solve the above problem, the process table can be divided into two so that the memory manager and the process manager will keep their respective parts.

3.2.3 The Object Model

The object model was first investigated in programming languages but has been applied to system design [Jon78]. Examples of it have been presented in the preceding chapter. Although different systems have different definitions of an object (e.g. some define it as an active entity, others treat it as passive), the underlying function is the same: an object provides an abstraction, very much like an abstract data type in programming languages. A system resource can be represented us-

ing an object. Instances can be created for each object type and they inherit the properties of the generic type. A set of operations is provided to manipulate the instances. Such operations maintain the sole access to the internal data structures of the object.

This methodology is attractive because it simplifies the design of complex systems considerably. Higher object types can be constructed from lower-level objects. For example, two file servers with different functionalities can be implemented based on a common generic object that provides the basic services. Of course, the generic object can be further decomposed into lower-level objects. In short, one can extend the system by providing more levels of abstraction, but unlike the traditional layered approach, each level is not necessarily based on the level directly below it.

Since each object has a well-defined interface to the outside world, it is easier to modify the system - changes to any object data structures affect only the procedures involved in that object. Reliability of the system is also improved: objects cannot corrupt one another since an object cannot modify the implementation details of another object.

3.2.4 Overall Architecture

Figure 3.2 depicts the overall architecture of FLEX. Consistent with the paradigm, the kernel is kept small to provide the basic mechanisms. Services are implemented outside the kernel as user-level objects. In particular, the file system, the scheduler and the memory manager are placed outside the kernel to enable those applications that might not want to use their services to bypass them. It is obvious that this architecture can accommodate alternative services of the same task. For example, one file server can be implemented specifically for database applications, and another can be implemented to emulate the UNIX file system.

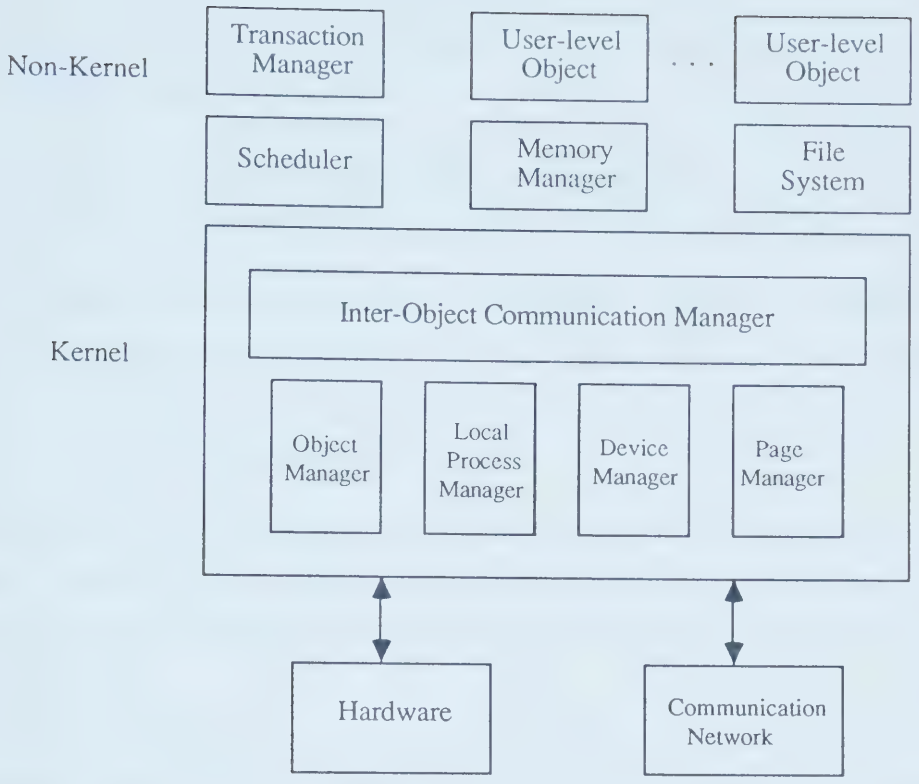

Figure 3.2: FLEX Architecture

3.3 Kernel Design

The FLEX kernel is designed to be compact. It implements five modules to provide low-level mechanisms for managing some system or virtual resources. The kernel modules are: Object Manager, Inter-Object Communication Manager, Page Manager, Device Manager and Local Process Manager. The general design of each of these will be discussed in the next section. The important features of the FLEX kernel are given below:

- The object types supported by the kernel are commonly needed in managing centralized and distributed computer systems as well as in distributed database management.

- A functionally identical kernel exists on each machine.
- Access to objects is uniform: both kernel and non-kernel objects are accessed via capabilities.
- Access to system services and other services provided by user applications are by invocation, which is handled by the Inter-Object Communication Manager.
- Message passing is used as the communication paradigm. The Inter-Object Communication Manager provides fully transparent access to remote objects.

3.3.1 Object Manager

An object-oriented system needs to provide mechanisms for users to create and to delete objects. In FLEX, such services are provided by the Object Manager(OM). A brief discussion of object management is given below; the details can be found in Chapter 4.

In FLEX, the primitive data types such as integers, reals and characters are not treated as objects. Therefore unlike other object-oriented systems such as Smalltalk [GR83], FLEX is not a pure object-oriented system. Its objects are typically large. We believe that this decision will significantly improve the performance of the system since there will be fewer objects to manage and less communication between objects for some primitive operations.

A FLEX object can be viewed as a three-tuple:

$$< \text{name, type, representation} >$$

where *name* is a system-wide unique identifier; *type* indicates the class that the object belongs to; and *representation* which consists of two parts: one is the resource that the object encapsulates and the other is the capabilities that the object holds for other objects. For each object, the OM maintains an object descriptor (OD) which contains information about the object.

The type field of each object instance contains the name of the type manager for that class. The type manager contains the codes of the operations that are defined for that class as part of its representation.

FLEX objects can be either active or passive. Active objects are those that have been invoked for service and are being serviced continually by the CPU. Passive objects, however, are stored in secondary storage and cannot be invoked. When an object is created, it is active. It may later become passive either as a result of scheduling decisions, or due to an explicit passivate command. Each active object is managed by an object coordinator which provides a number of services. The details will be discussed in the following chapters.

3.3.2 Inter-Object Communication Manager

The Inter-Object Communication Manager (IOCM) handles all communications between objects, whether they are local or remote. To provide a uniform semantics, calls to any kernel object go through the IOCM. Object invocation in FLEX is message-based and the format of the message is:

<capability, operation, parameters>

The format of a capability is described in Chapter 5; for now, one can regard it as a token that gives the holder permission to invoke an object. The operation specifies which routine that is defined for the object is to be invoked and the parameters are the parameter lists for the routine.

FLEX supports two types of invocations: synchronous and asynchronous. In synchronous invocation, the sender object blocks until a reply is received. Synchronous invocation is useful in defining remote procedure calls at the programming language level. Asynchronous invocation permits the sender to continue operation before a reply is received. Thus it maximizes parallelism but tends to make programming tricky and hard to debug.

To provide the two different forms of invocation, IOCM provides both blocking and non-blocking "send". The IOCM uses the mailbox concept. For every object,

conceptually there exists a mailbox where messages can be deposited.

3.3.3 Page Manager

The Page Manager continues the practice of the separation of mechanisms and policies principle. In Chapter One, we have outlined the difficulties of implementing a DBMS on top of the conventional virtual memory operating system. To overcome the problems, the Page Manager will provide the *mechanisms* to support the implementation of a virtual memory system as well as the implementation of the various buffer management strategies (e.g. the hot set [SS86] and the query locality set model [CD86]), both of which have been shown to be more appropriate for database management.

It should be pointed out that this is a service for which it is very difficult to enforce the rule of separating policies from mechanisms. Any decision to place some component of the memory and buffer manager outside the kernel has considerable performance implications. What we have chosen to do, in the FLEX kernel, is to provide the minimal facilities for memory and buffer management and leave the implementation of the policy decisions to modules outside the kernel. The kernel implements the *page* object. The two fundamental operations that need to be provided by the Page Manager are (1) to load a page identified by its number, and (2) to unload a page.

3.3.4 Device Manager

The Device Manager manages all device drivers. Typically each device driver handles one device type or a class of related devices (e.g. a terminal driver might handle several brands of terminal). A device-driver implements the code that is necessary to interface with the device. Furthermore, it accepts requests from the device-independent software above it. The device-independent software can be embedded in the file system and it performs I/O functions that are common to all devices.

3.3.5 Local Process Manager

The Local Process Manager implements the process model. A process is a unit that may be scheduled for execution. The Local Process Manager performs the following functions related to process and processor management:

- provides mechanism to create and destroy processes.
- performs process dispatching which involves giving control of the CPU to the process that is selected for execution.
- provides interprocess communication and synchronization primitives, and
- implements methods to manage deadlocks.

The operations defined by the Local Process Manager can only be invoked by objects that are local to the machine. Remote process creation is not supported. Process dispatching is further described in Chapter 6.

3.4 Non-Kernel Design

The previous section provides an overview of the FLEX kernel functionality. There are a number of OS functions that are left to be implemented at the user level. In addition, little DBMS functionality is implemented inside the kernel. In this section, we will briefly describe the remaining OS functions that are to be implemented on top of the FLEX kernel.

The OS functions that are implemented as user-level modules include the scheduler, the memory manager, the file server and the user shell. The scheduler makes all scheduling decisions and is described in Chapter 6. The memory manager implements the policies that govern the use of main memory. The file server implements operations to operate on file objects and makes use of facilities from the page manager and the device manager. It can also implement its own memory cache buffering scheme that facilitates its secondary storage access. The shell implements the command interpreter to interface the users with the operating system.

Developing a distributed or centralized DBMS on top of the FLEX kernel should be relatively easy. The kernel provides essential services such as transparent access to distributed objects, mechanisms for manipulating memory pages, uniform naming and access control facilities. However, most of the other OS facilities are pulled out of the kernel and they will not get in the way of the DBMS. For example, the DBMS can implement its own buffer manager on top of the FLEX kernel that is independent of the OS memory manager. Nevertheless, the suitability of the FLEX architecture for database applications is subject to verification. As a first step, the kernel structures are designed and implemented. The following chapters describe in detail some of these designs.

Chapter 4

Object Manager

By means of the object model, the OS can be described as a set of object types, each of which can represent either a logical or a physical resource. Processes, semaphores and mailboxes are examples of logical types whereas I/O devices and memory addresses are examples of physical types that may be supported by the system. The object types that are directly supported by the kernel are often called *primitive type objects* [Lin76]. They are used in creating new object types which are sometimes called *extended type objects*. The Object Manager (OM) is responsible for managing these object types as well as for providing a uniform semantics for declaring new types. The latter is done by following a type hierarchy.

It is important to indicate, at this point, the role of objects in FLEX. It is possible to achieve the goals of FLEX without using the object-oriented design paradigm. A minimal kernel which implements only the necessary mechanisms would provide the necessary flexibility to implement the D-DBMS functions within its own address space. Object-orientation has been incorporated into FLEX for two fundamental reasons: (1) to make use of the encapsulation opportunities provided by object-oriented system design, and (2) to provide a base for the object-oriented database research that some members of the Distributed Database Systems Research Laboratory are involved in.

In this chapter, we will discuss the object base that has been incorporated into the FLEX kernel. We will first discuss the object model (Section 4.1) and then

present the management of objects (Section 4.2).

4.1 Object Model

The precise object model that should be supported by an operating system kernel is a current research issue that is being investigated by the members of the Distributed Database Systems Research Laboratory. The details of the model are dependent not only on the requirements of the operating system, but also on those of the DBMS. Therefore, in this initial prototype, we have chosen to implement a simple object model. This will be changed in the next version of the prototype.

We have already mentioned that not all system entities are objects in FLEX. The primitive data types (e.g. the integers, reals, booleans and characters) are not treated as objects. All objects in FLEX are a three tuple:

<name,type,representation>

The *name* field uniquely identifies the object and is described in Sec 4.2.1. The *type* field of an object can have two different meanings, depending on what the object is. If the object is an instance of a type, then this field contains the name of the type manager defined for that object. For example, objects can be of type FILE, PROCESS etc. The other case is when the object actually defines the Type. In this case, its *type* field contains the string "type manager". The "type manager" object contains the code for the operations that are defined for that Type. Each instance of the Type *inherits* the properties of the generic type. There is no need to duplicate the code of the operations that manipulate the type for each object instance.

As mentioned in the last chapter, the *representation* field contains two parts: the resource and the capability. For an object that is of type "type manager", the resource contains the code and any data structures that define the Type. For an object that is an instance of a type, the resource encapsulates the code or data that is local to the object. The capability part contains the list of capabilities that the object holds for another object. For example, if the capability part of an object

contains a capability for the Page Object, the object can invoke the operations that are defined in the Page Object provided that the capability has the necessary rights. Capabilities will be discussed in the next chapter.

As an example, consider an OS that supports objects of type FILE. Operations such as Read, Write, Open, Close can be defined for that object type. A user with the necessary rights, may create two files, say File1, File2 which are instances of type FILE. The operations defined for FILE can be applied to these instances. This situation is depicted in Figure 4.1. This concept of a type hierarchy has been investigated in programming languages like Simula [DMN68] and applied to OS design as in Hydra.

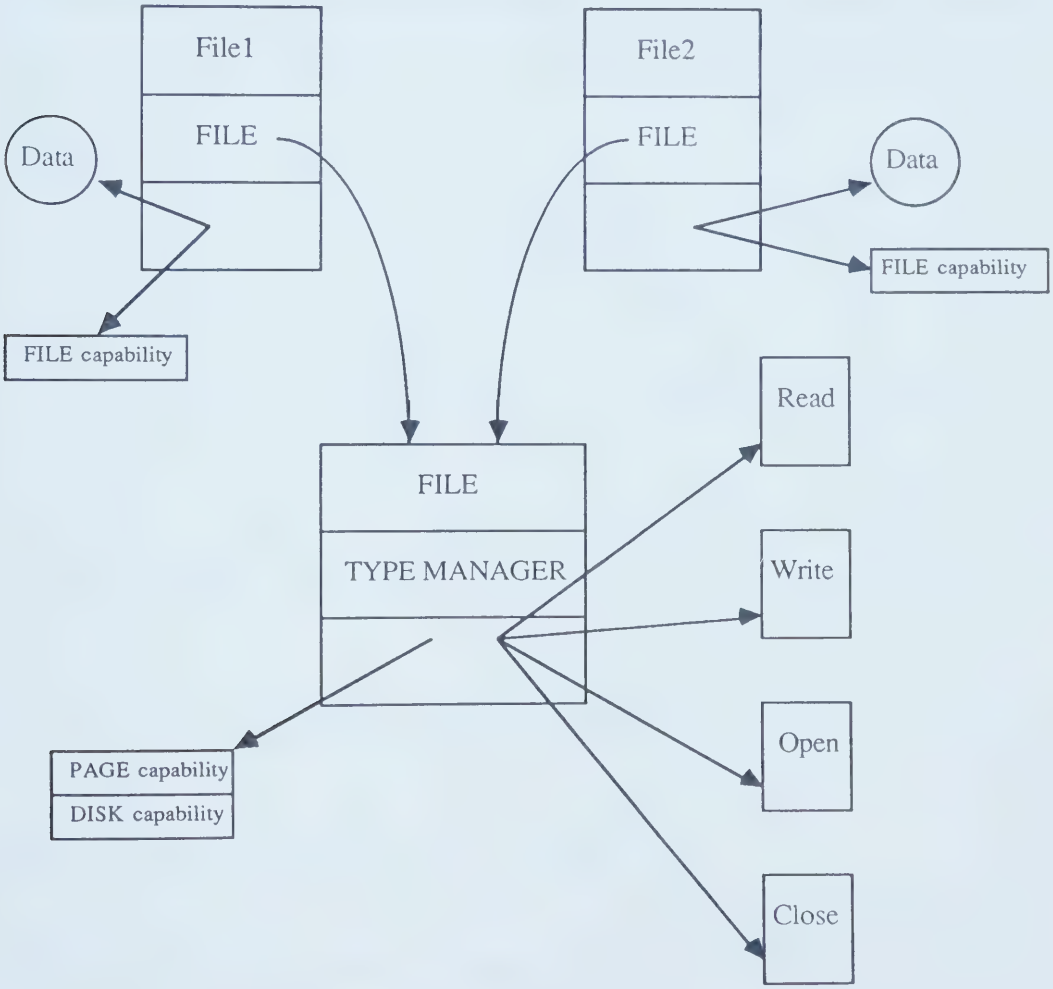

Figure 4.1: Example Object Structure

4.1.1 Type Hierarchy

In a real object-oriented system, objects belong to *classes*, which correspond to *type* in FLEX. Object classes can then inherit properties from other classes. The inheritance mechanism typically works by specializing a previously defined class(es) to perform a slightly different function.

A similar hierarchical organization of types is supported in FLEX. As demonstrated in Figure 4.1, each object is an instance of a type which are represented by type managers that contain the codes of the operations that are defined over that type. The type managers themselves are instances of an object of type "object" that is implemented inside the kernel (similar to [WLH81]). This forms a hierarchy of objects whose root is in the kernel. An example hierarchy is depicted in Figure 4.2.

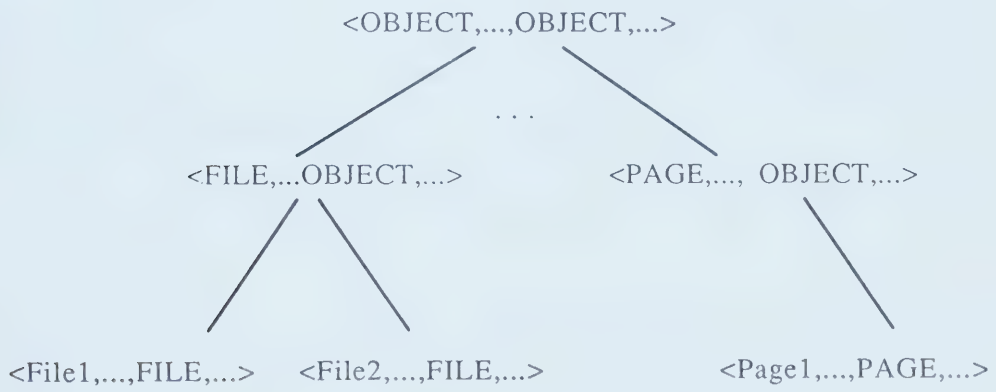

Figure 4.2: Object Hierarchy

4.1.2 Active Versus Passive Objects

In Chapter 3, we mentioned that all FLEX objects can be either active or passive. The structure discussed above and shown in Figure 4.1 pertains to passive objects. The structure of active objects is different and is depicted in Figure 4.3.

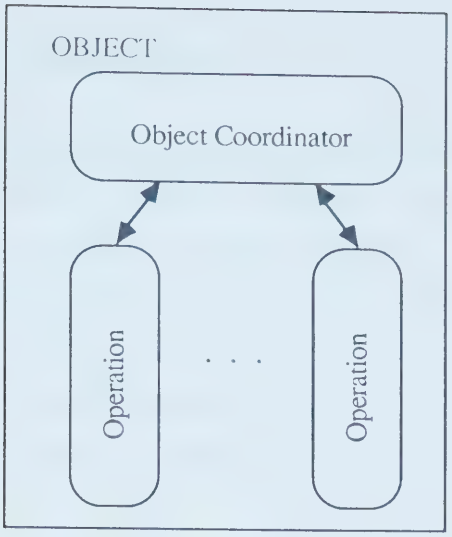

Figure 4.3: Active Object Structure

Each active object consists of an *object coordinator* which is responsible for providing a number of services on behalf of the object. The object coordinator is a generic piece of code that is linked together with an object and is executed when an object is created or when a passive object is activated. The object coordinator has the following responsibilities related to invocation:

1. Message resolution: Determining from an incoming message which operation is being addressed.

2. Rights verification: Determining whether the invoker actually has the right to perform that operation on this object. The details will be described in the next chapter.

3. Operation activation: Activating the procedure that is being invoked by creating a scheduable unit and ask the scheduler to schedule it for execution. The details of the scheduler will be described in Chapter 6.

Object invocation in FLEX is performed by sending a message to that object. The form of the message has been described in Chapter 3. The details for the communication mechanism can be found in [Li88].

4.2 Object Management

The object manager is responsible for creating, destroying and for the overall management of objects. The object manager will be a fundamental research issue in FLEX since it can have an effect on the DBMS operations by controlling the placement of objects in secondary storage and by employing different garbage collection mechanisms. Since this module is the subject of ongoing research, we have chosen to implement a simple object manager at the present time. Although simple, the object manager provides adequate support for object naming, object addressing and object manipulation.

4.2.1 Object Naming

The naming scheme that is employed in FLEX has the following features:

1. **Sharing of objects**. Multiple accesses to an object from other objects are permitted. This is the result of using capabilities for accessing all resources in the system.

2. **Replication support**. Objects can be replicated. If there are replicas of a given object, it is possible to access any or a specific copy of that object.

3. **Fully transparent access**. All objects can be invoked or addressed in the same way, regardless of their location.

To support the above features, FLEX objects have two names: an internal name and an external name. Internal names are fixed length binary numbers, generated by the system. External names are user-defined character strings. Creation of any object is caused by invoking the creation routines (Sec 1.2.4). The caller supplies a character string which denotes the name that he wants to call the object. Note that the character string need not be unique (e.g. two users can create two different objects called **tmp**). To distinguish between objects with the same name, FLEX generates an internal name for each newly created object. The internal name is guaranteed to be unique and FLEX operates on all objects using their internal names.

As we will see, this will provide transparency and facilitates efficient manipulation of objects within the FLEX kernel.

The internal name in the current prototype is a 32-bit value which can be adjusted in a future implementation. Given this length, the internal names can range from -2^{31} to $2^{31}-1$. A fixed number of internal names are reserved for system use (e.g. to identify users in the system) and the number can be determined at system initialization time. The remaining names can be used to name objects with the requirement that the range of internal names on each machine be fixed. As we will see shortly, this is useful for providing a clean semantics for handling objects and replicas.

All the object replicas will have the same internal and external names as the original. This is necessary because any update has to be performed on all copies. If the replicas have the same names, then one can update all the copies using a broadcast mechanism by specifying the internal name of the object.

To generate a unique internal name, a global counter can be used. It is incremented and returned every time a new name is needed. However, like any global resource, a global counter has serious performance drawbacks. A better way is for each machine to keep its own local counter which is incremented every time a new name is requested on that machine. The problem then is to ensure that the internal names generated are still system-wide unique. For example, if an object with internal name 123 is generated on machine A, then any other object with the same internal name cannot be generated elsewhere in the system.

One way to solve the problem is to reserve some bits in the name to denote the site from which an object is created. In this case, **object 123** from machine A can be separately identified from **object 123** on machine B. We rejected this approach because its semantics are unclear. In a distributed system, objects can move from machine to machine. When **object 123** is moved from machine A to machine B, **object 123** site bits no longer tell us which site it is in. We cannot modify the site bits to denote machine B because, in that case, an object with identifier 123 might possibly be generated on machine B already, and hence we have two objects with

the same internal name.

Our approach is to have a fixed valid range of internal names on each machine. For example, names 1-10000 can be generated on machine A; names 20000-35000 can be generated on machine B, etc. In this way, moving objects from machine to machine will not cause any problem. We do realize, however, that naming in a distributed system is a significant problem which requires further investigation.

After an object is created, a capability is returned to the creator. The capability contains the internal name of the object. All future references to that object are done via the capability, i.e. the capability alone will be the holder's key to the object. Even when the object is moved to another site or to another directory, the creator need not be aware of it. Hence transparency is achieved.

The creation of an object causes the creator to automatically receive a capability for access to that object. Others can access the object if they have a capability for it. To obtain access to any object, a kernel message **ObtainCap** can be sent. The caller will supply the name of the object (e.g. file.c, troff) for which he or she wishes to obtain a capability for. This is the only situation where the caller might need to supply information about the location of the object; if the OM cannot resolve the name unambiguously, then the user will be asked for further information. Note that this is a reasonable expectation because in any operating system environment, if one wishes to use a resource that belongs to the system or to another user, the existence and the owner of the resource need to be known by the requester. Furthermore, pathname specification facilities similar to UNIX can be provided at the user shell to facilitates this process.

4.2.2 Object Addressing

Capability provides reference to objects. On early capability systems such as the Chicago Magic Number Computer [Fab74], a base/limit field is placed in the capability. The field is updated whenever the object moves between primary and secondary memory. This causes substantial overhead especially when there are more than one capabilities/references to the object. A common solution to this

problem is to have capabilities point to the objects indirectly through some kind of mapping table. Each mapping table entry is usually called an object descriptor and contains information about an object. In particular, it stores a disk address if the object is in secondary memory and it may contain a pointer to the object if it is in primary memory. Then when the object is moved, only the mapping table entry has to be changed. Figure 4.4 depicts the situation. Examples of this approach include the Hydra Global Symbol Table [WLH81] and the Intel Object Table [Org83].

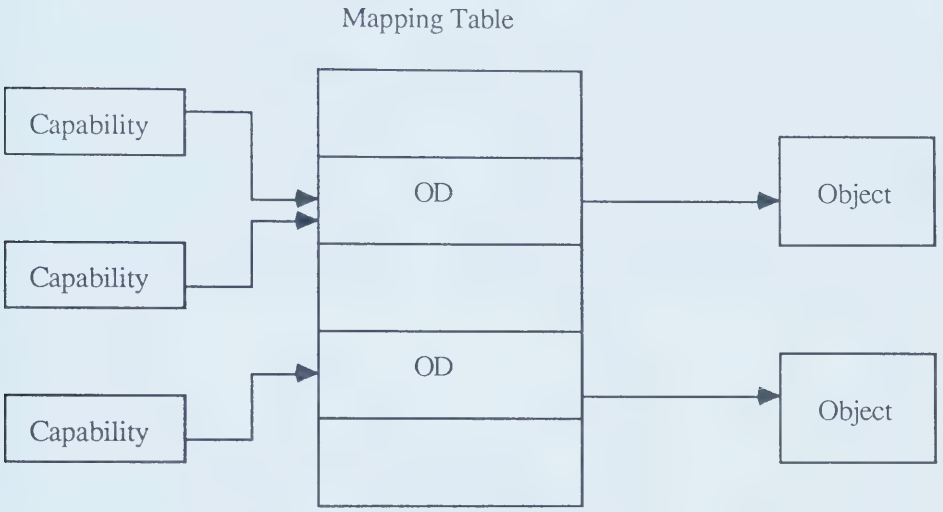

Figure 4.4: Object Addressing

In FLEX, a single level directory exists at each site. It includes all the local object descriptors. The directory maintains two Object Tables (OT), both of which are managed using hashing. One of the Object Tables is for access by internal name (*Internal Object Table - IOT*) while the other one is for access by external name (*External Object Table - EOT*). Given an internal/external name, it is hashed into the corresponding table which points to a set of object descriptors. The structure of an object descriptor is described in the next section. The object descriptors are maintained in a sorted fashion in both of the tables. In the IOT, the object descriptors are sorted on the internal name, whereas in the EOT, the object descriptors are sorted on the external name. The structure of the object directory is given in

Figure 4.5.

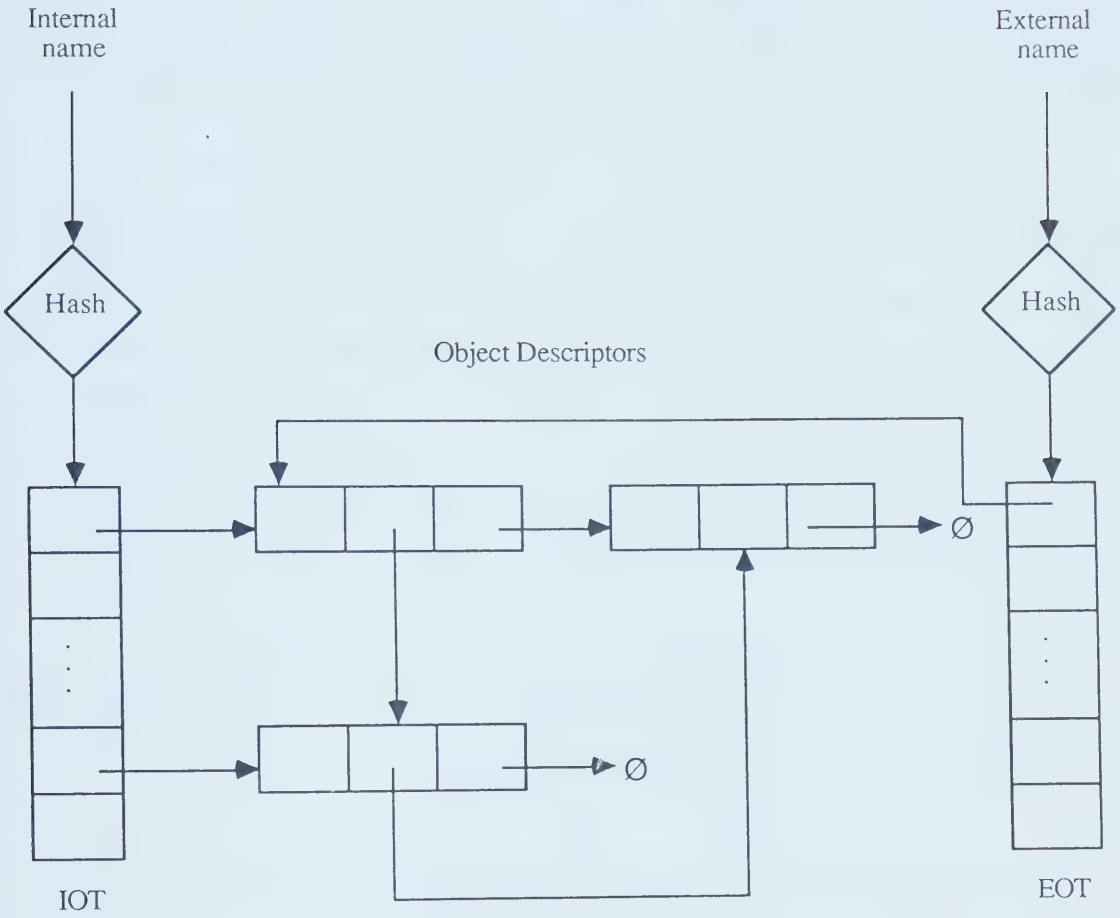

Figure 4.5: Object Directory

Both the IOT and the EOT are maintained in main memory but the object descriptors are stored in secondary memory. This is a reasonable approach since the object directory is accessed only when a *passive* object is being invoked. The object descriptors for any active objects are stored in the object directory as well as in a main memory structure within the inter-object communication manager. This main memory structure is called the *Local Active Object Table* (*LAOT*) and has the same structure as the object directory.

The sizes of the IOT and the EOT are equal and this size is fixed at system initialization time, subject to the available memory on each particular machine. To facilitate fast access, it is of course preferable to increase the size of the IOT and the EOT so that collisions in a hash bucket are minimized.

4.2.3 Object Descriptor

When an object is created, an object descriptor (OD) is also created for it at that site. Information about each object is stored in the object descriptor which is maintained by the kernel. Note that if the object is replicated, a descriptor for it exists at the site where it is first generated. This is to simplify the maintenance of replicas since it is easier to enforce concurrency control mechanisms if the number of all the existing copies of an object is known at the originating site.

The format of an OD is given in Figure 4.6. The meanings of each field are as follows:

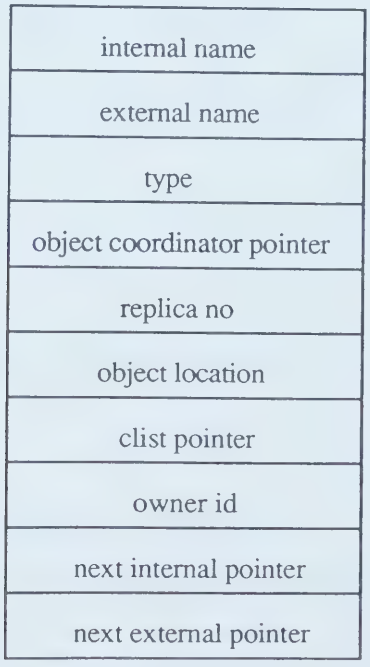

Figure 4.6: Structure of an Object Descriptor

1. internal name - the integer that uniquely identifies the object.

2. external name - a character string that the user choose to name the object.

3. type - a character string describing the type of the object. The string can either be "Type_Manager" or the name of the *type manager* that defines the object type for the object.

4. object coordinator pointer - an integer that stores the process identifier for the object coordinator of the object.

5. replica no - an integer describing the number of replicas that exist for the object.

6. object location - a disk address/pointer to the object. This field is updated if the object is moved to another location.

7. c-list pointer - a disk address/pointer to the capability list held for the object. The details will be deferred to Chapter 5.

8. owner id - an integer that identifies the owner of the object. The details will be deferred to Chapter 5.

9. next internal pointer - disk address/pointer to the next OD based on internal name.

10. next external pointer - disk address/pointer to the next OD based on external name.

4.2.4 Object Manager operations

Currently, the OM supports the following seven operations:

1. **Createobj**: creates a new object type. A capability for the new object type is returned to its creator.

2. **Createinst**: creates an instance of the specified type. A capability for the new object instance is returned to the creator.

3. **Createuser**: creates a new user object descriptor, and returns a capability for it to the user. The details for this is described in Chapter 5.

4. **Destroy**: destroys an object type or an object instance. The capabilities held for the object will also be deleted.

5. **Createreplica**: creates a replica of an object instance. It also creates a new descriptor for the object and stores it in the local directory.

6. **Place**: places an object at a specific place. This facilitates the placement of objects and/or their replicas at other sites. It causes the invocation of the IOCM to transmit the object and the corresponding descriptor to the OM at the destination site.

7. **Search**: goes through the directory looking for the descriptor of a given object. It will return information about the object such as its owner, its size etc.

Chapter 5

Access Control and Protection

Conventional operating systems support two modes of execution: privileged (or kernel) mode and non-privileged (or user) mode. Processes can execute in either the user or the kernel mode but a process is only permitted to access kernel structures when it is executing in the kernel mode. To switch from the user to the kernel mode, a system call or an interrupt has to be made. Such a structure leads to a monolithic design: any service that requires protection must reside in the kernel. The result is a large and complex system that is hard to extend. FLEX's attempt to move traditional OS services out of the kernel requires a different protection mechanism. For example, services at the user level must be protected from illegal access or modification by other user level objects.

In FLEX, an object is the unit of protection because it is the abstracted notion of an arbitrary resource and is the only entity known in the system. Access to all objects is done via capabilities. A capability serves two purposes: it identifies the object and it indicates the rights the holder has for the object. When an object is created, a capability is also created. This new capability includes all the access rights defined for the object and is the owner's key to the object. Sharing of an object is done by having multiple capabilities to the object. These capabilities might have a different set of access rights so that different holders can have different privileges to the object.

5.1 Protection of Capabilities

5.1.1 Background

Since access to objects is derived from the possession of capabilities, we must ensure that users cannot forge capabilities or modify them illegally. Traditionally there are two ways to maintain the integrity of capabilities: tagged memory and partitioned memory [Fab74]. On a system with tagged memory, each memory word has one or more tag bits associated with it. The tag tells whether the word contains a capability or not. The setting and testing of the tag bits is done by the hardware each time an access is made. Arithmetic and ordinary instructions can only be applied to words that are not tagged as a capability. Addressing, on the other hand, is only allowed on words that are tagged as a capability. Examples of systems that use the tagged approach include the Burroughs B6700, the Rice Computer and the IBM System 38.

When partitioned memory is used, all memory segments are divided into two classes: capability segments and data segments. Capability segments may contain only capabilities and they are often called capability lists or C-lists. Data segments may not contain capabilities. The capabilities are kept inside the kernel and can only be modified by the OS. Examples of this approach include the Plessey System 250, Hydra, Intel 432, etc.

The disadvantage of the tagged approach is the space consumed by the extra tag bits. Furthermore, it requires direct hardware support and cannot be used on conventional non-capability-based machines. The disadvantage of the partitioned approach is that since most objects have both data and capabilities, two segments are required instead of just one as in the tagged approach. This tends to complicate operations because they have to deal with two different segments.

A more recent approach suggested by Tanenbaum and Mullender [TMvR86, MT84] is to keep the C-list in user space but encrypt it with a key unknown to the user. The basic idea is that when an object is created, the server responsible for its creation will pick a random number and store the number inside its table

as well as in the newly-formed capability. When the client uses the capability to reference the object, the server looks up the object and then compares the stored random number with the one contained in the capability. If the two matches, the capability is considered to be valid. Further protection mechanisms can be designed on top of this scheme. For example, the random number can be used as the encryption/decryption key; the access rights field can be protected using one-way functions.

5.1.2 FLEX Approach

Since FLEX is built on conventional machines with no special hardware support, the C-list approach is used. The capabilities are kept inside the kernel. The way of referencing a capability from user space is quite novel, and as we will demonstrate shortly, the approach solves some of the problems intrinsic in capability systems.

The structure of a FLEX capability is given in Figure 5.1. This is the format that is stored in the kernel. The copy that is given to the user has a different format and is shown in Figure 5.2. To make the distinction clear, we refer to the capability stored in the kernel as **RealCap** and the capability given to the user as **UserCap**.

Figure 5.1: FLEX Capability

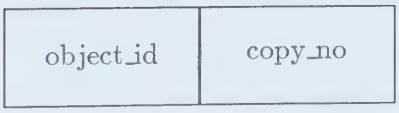

Figure 5.2: User Capability

When an object is created, both the **RealCap** and the **UserCap** are created. Within the kernel, the **RealCap** will be kept in the object directory, which is

maintained by the OM, as well as in the LAOT, which is maintained by the IOCM. Recall that when an object is created, a new object descriptor is created and it goes into both the object directory and the LAOT. In the first case, the object descriptor is stored in secondary memory whereas in the latter case, it is stored in main memory. In both cases, the object descriptor has a C-list field which stores either a disk address or a pointer to the RealCap that is generated for the object. Figure 5.3 depicts the situation.

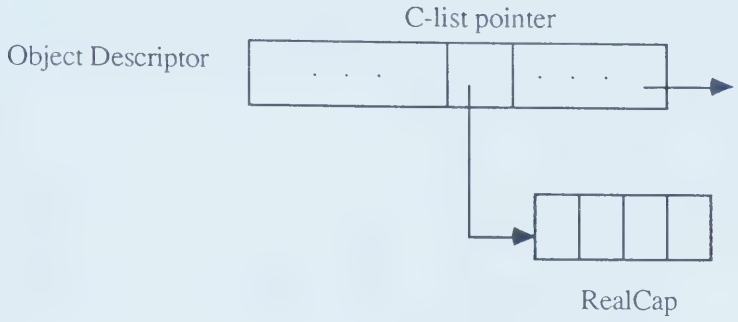

Figure 5.3: C-list in FLEX

As we have seen, an object is uniquely identified by its *internal name* which is generated by the system when the object is created. This *internal name* is stored in the object descriptor as well as in the *object_id* field in the **UserCap**. The **UserCap** is returned to the creator so that he can use it to make further references to the object.

In a capability-based system, object sharing is achieved by allowing multiple capabilities to the same object. Such capabilities can have different access rights. The system provides operations to generate these capabilities. To keep track of the multiple capabilities, we have the field *copy_no* in both the **RealCap** and the **UserCap**. This field stores a unique integer that is generated by the system when a capability is created. When an object is referenced using a **UserCap**, the system looks up the corresponding object descriptor using the *object_id* field. Then the corresponding capability is located by using the *copy_no* from the UserCap. An

example is given in Figure 5.4.

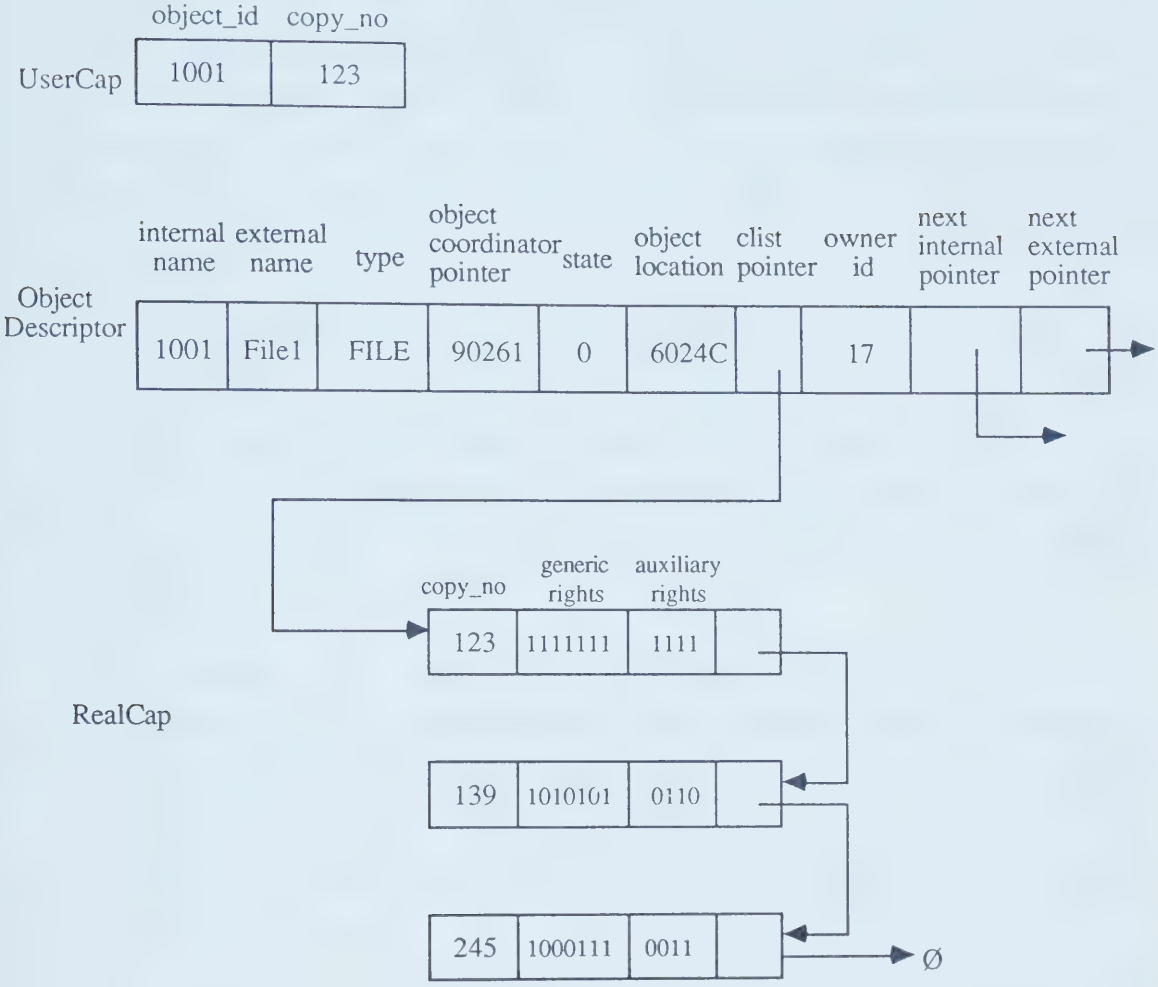

Figure 5.4: Example showing object sharing with different capabilities

In this diagram, the object descriptor for the object **File1** is shown. The *internal name* for the object is **1001**. Three capabilities have been generated for the object and each bears different access rights. The capabilities are maintained as a list sorted on *copy_no* inside the kernel. The object can be accessed using the **UserCaps** that are returned, one of them is shown in the diagram.

The *rights* field in the **RealCap** is a bit map where each bit defines one operation that can be performed on that object. This convention simplifies rights restriction

and rights checking and makes the related operations efficient. The access rights are divided into two fields: a *generic* rights field and an *auxiliary* rights field. The generic rights is defined by the system. It is the set of rights that can be applied to any FLEX object. Auxiliary rights, on the other hand, are type-specific and are defined by the creator of a new object type. It is interpreted by the operations that operate on that specific object type. The generic rights that are supported by the system are given below:

1. **CreateInstRts** - This right allows an instance of an object to be created. The **Createinst** routine in the Object Manager takes a capability that contains this right, creates an instance of the object and returns a capability for the object. The new capability has the same rights as the capability supplied for calling the routine. This right can never be amplified. Amplification will be described in Section 5.2.

2. **CopyRts** - This right allows a new capability for the same object to be created. That is, it allows object sharing. However there is a problem of preventing others from adding more rights to a capability once the capability is given to them. To limit such unauthorized amplification, there is the RestrictRts which we will describe next. Like **CreateInstRts**, this right can never be amplified.

3. **RestrictRts** - This right, if present, prohibits one from altering any rights of a capability. Note that **RestrictRts** cannot be removed once it is set. **RestrictRts** is obviously very restrictive and one normally does not want to set it.

4. **ReadDataRts** - This right allows data to be read from the data part of the object addressed by the capability.

5. **ReadCapaRts** - This right allows a capability to be read from the capability part of the object addressed by the capability.

6. **WriteDataRts** - This right allows data to be written to the data part of the object addressed by the capability.

7. **WriteCapaRts** - This right allows a capability to be written to the capability part of the object addressed by the capability.

5.1.3 Advantages of FLEX Approach

It has been observed that revoking access to an object is quite difficult in a capability-based system [Lev84,Tan87]. Once a capability is granted, it is hard for the system to take it back since capabilities are stored in C-lists that may be scattered all over the disk. For example, in Hydra, capabilities that a user holds for other objects are stored in the user's own LNS; capabilities that an object holds for other objects are stored in that object's C-list part. The lack of global knowledge about capabilities makes revocation quite complicated.

The FLEX capability management scheme, however, makes revocation quite simple. Observe that all the capabilities for an object are kept track of by using the C-list field of the object descriptor. Therefore, it is fairly simple to take back capabilities for any object after they are given out. For example, an authorized party can revoke somebody's access to an object by simply deleting his capability. Any further reference to the object using the revoked **UserCap** will not be granted since the **RealCap** no longer exists. Another advantage is that when an object is destroyed, all the capabilities for that object can be deleted easily. Thus there is no need to perform garbage collection on capabilities.

5.2 Protection of Objects

In FLEX, protection of access to all object types is provided uniformly, i.e. for the purpose of protection, there is no distinction between kernel and non-kernel objects. There are two levels of control with respect to protecting objects, the first is over the creation and the second is over the object access.

We saw in the last chapter that object creation is handled by the OM. Practically every user can create a new object type. Control over the creation of instance objects is governed by the possession of a capability for the type object and the capability must have **CreateInstRts** set. To obtain such a capability, a kernel message **ObtainCap** can be sent. The set of kernel calls that handle capabilities is given in Section 5.4.

Control over access to a previously created object is governed by the possession of a capability with the appropriate rights. To build a secure system, the *principle of least privilege* has to be observed [Lin76,Jon78]. The basic idea is that the set of access rights that a program has for other objects (i.e. its protection domain) should be as small as possible. For example, the domain for which an operation is to be performed should contain only those capabilities that are required to perform that operation. Furthermore, the capabilities should contain the minimal set of rights.

As an example, consider a device manager which handles the disk. The device manager alone knows about sectors, tracks, arm motions and other mechanics of making the disk work properly. Any software that initiates a request does not need to know how the request is represented or carried out and can take an abstract view of it. To make a request, all the software needs is a capability for the device manager with the rights set for the particular request.

A problem arises when a type manager module needs to manipulate the internal representation directly in order to carry out a request. Continuing with the above example, assume that a user has a capability for the device manager with read rights and issues a read request. The user capability need not contain the right to address the disk. However the device manager needs to address the disk before it can perform the request. In this case, an operation called *amplification* [JW75] is needed to transform the capability. The procedure implementing the type, in this case, the device manager, will verify that the user supplied capability is a valid capability for the device manager and then determine which rights it is missing. A new capability with the missing rights will be manufactured and used to perform

the request.

The ability of a procedure to amplify certain access rights means that the access rights of a capability can be increased as the capability propagates around the system. Theoretically, the procedures implementing a particular type can use rights amplification to reinstate those rights that are restricted by the caller of the procedure. Therefore some rules must be set to limit the unwarranted propagation of information. In FLEX, the **CreateInstRts** and the **CopyRts** can never be gained through amplification. A capability lacking **CreateInstRts** represents an intention to prevent new instances being created. Obviously we do not want any user to create new instances of the scheduler or the file system. Likewise, a capability lacking **CopyRts** represents an intention to prevent object sharing with a third party. Finally, the **RestrictRts** in FLEX disallows any sort of amplification. Normally the invoker of a type object would not set it because he would presume the operation is reliable. If he does not trust the procedures, he should probably not use them.

5.2.1 Domain Switching

Large programs usually need to access many objects. However, a routine within it typically only access a small number of objects. Therefore, to keep the protection domain small, a large program should execute in many different protection domains and constantly switch between these domains during its execution. Advocates for small protection domains maintain that a domain change should be performed on every procedure call [SS75]. Although there are machines that do perform a domain switch at every procedure call, (e.g. Intel 432 [Org83], Plessey System 250), we argue that it may not be desirable for the following reasons:

- It can be inefficient. The performance issue can be further emphasized as follows:

    1. It has been noted that procedure calls are among the most frequently executed instructions, accounting for 12% to 25% [GC86] of program code.

Linden has observed that a procedure call that performs a domain switch typically requires a millisecond or more. Although this statistic was collected in the seventies and technology has improved since then, there is still reason to believe that a domain switch for every procedure call can lead to a serious performance bottleneck with unnecessary complications. This is validated by performance studies on the Intel 432 [GC86]. With all the necessary hardware support, performing a domain switch on every procedure call has been observed to run slowly. Therefore, it would be unreasonably expensive if it were to run on a conventional machine.

2. A domain switch causes allocation of a new object (e.g. LNS in Hydra, context object in Intel432). The creation of an object is normally expensive and it involves allocating memory for the object, adding it to some sort of object table for access, deallocating and deleting the objects when the procedure returns etc. Of course one can argue that such objects can be preallocated, as in the case with Intel, but there is no guarantee that such objects will not be exhausted (e.g. in highly recursive programs) and in this case, we will fall into the same problem.

- It is inflexible.

1. A domain switch on each procedure call results in all the procedure calls being protected. Having only protected procedure calls can be inflexible. If the system provides a choice between protected and unprotected procedures, then applications that require maximal protection (e.g. a password subsystem) can choose the former one whereas applications that require maximal performance (e.g. context switching) can choose the latter one.
2. One can view recursion as a form of iteration that involves procedure calls. In this sense such calls would not involve a domain change, i.e. the set of objects accessible to a recursive call is the same. Recall that objects in FLEX are not small objects like integers or reals. Therefore,

forcing a domain switch on procedure calls that do not involve a change in domain is both inflexible and unnecessary.

We believe that neither a domain switch on every procedure call nor a domain switch only when a system call is made is appropriate. In FLEX, a domain switch is performed by default when a call is made across object boundaries. Otherwise domain switching will be performed only when a call is made to a protected object.

5.2.2 Protection Enforcement Using Object Coordinator

We mentioned in the last chapter that an object coordinator (OC) exists for every active object in FLEX. The OC provides a number of services on behalf of an object. As we will see shortly, a major responsibility of the OC is to enforce the protection mechanisms.

For a newly created object, the OC performs any necessary object initialization by calling an *initialize* routine that is written by the object's creator. Furthermore it sets up two tables for the protection mechanism. The first one is the Operation Table and the second one is the Access Table.

The creator of an object has to tell the OC the access rights for each operation using a specified format. The OC records this information in its Operation Table. The format for each entry is given in Figure 5.5.

Figure 5.5: Structure of the Operation Table

The *operation_name* is a character string that stores the name of the operation; *generic* and *auxiliary rights* are unsigned integers that store the access rights for the corresponding operation; *next* is a pointer to the next record. The Operation Table is maintained as a linked list sorted on *operation_name*.

The Access Table will keep track of those objects that are allowed to access this object. The format for each entry is given in Figure 5.6.

Figure 5.6: Structure of the Access Table

The *internal_name* is the integer that uniquely identifies the object. As before, the *generic* and *auxiliary rights* store the access rights and *next* is a pointer to the next record. However, the Access Table will be empty initially, i.e. a newly created object can only be accessed by its creator. Permission for others to access the object has to be added at a later stage using a kernel message **AlterRts** which is described in Section 5.4.

After performing the above setups, the OC goes into a wait state, waiting for an incoming message. The message will be deposited into the OC's mailbox by the IOCM. Currently there are three message types: **Alter**, **Obtain** and **Invoke**. The structure of the mailbox as well as the message format are described in [Li88].

Alter

This message is sent by the owner of the object to inform the OC to change the access rights for a particular user or object. Thus, along with it, the internal identifier, the generic and the auxiliary rights will be deposited into the mailbox. The OC will retrieve the information and store it into the Access Table.

Obtain

This message is sent by any user or object to request permission to access the object. Along with it, the internal identifier of the user or the object is sent. The OC will lookup the Access Table and if the identifier is found, the corresponding generic and auxiliary rights will be returned. If the identifier is not found, an error code is returned.

Invoke

This message can be sent by any object to invoke a particular operation within the object. The capability for invoking the object, the operation name and the parameters are deposited into the mailbox. Recall from the last chapter that invocation proceeds in three steps, namely, message resolution, rights verification and operation activation. To verify that the invoker has the necessary rights, the OC searches through its Operation Table. If the capability has sufficient rights, the OC proceeds to step 2. Otherwise, the OC will try to perform amplification according to the rules discussed before. If amplification succeeds, the OC proceeds to step 2. Otherwise, an error code is returned.

5.3 Protection of Users

A user may need to identify himself when sending some kernel messages. Examples of such are **ObtainCap** and **AlterRts**. To identify oneself, the user has to supply a capability. This capability is given to him when he first registers in the system. In fact, a user is just another object in the system, possessing two names: an external one, which will be the user signon identifier and an internal one, which is generated by the system. Of course the external name has to be unique for user objects.

To preserve uniform semantics, the capability given to the user has to have the same format as the **UserCap** we described earlier. The first field *object_id* will contain the internal name, but since only one copy of this capability will ever be issued for each user, the second field *copy_no* is empty. We use it to perform some additional security checks.

Because the internal name of the user is generated from a set of reserved numbers which has a relatively small range (e.g. 1-1000), a malicious user might try to disguise himself as another person by changing the internal name from the *object_id* field in his capability. For example, if user A's internal name is 10, then he might change it to 350 which may happen to be another user in the system.[1] To prevent

[1]The chance of forging capabilities for other objects also exist but the chance is much smaller

this, we encrypt the internal name using a key known only to the kernel. The encrypted form is stored in the second field of the capability. For those kernel services that require verification of a user, the second field is decrypted using the key and checked against the first field. Note that because the first field of the capability stores the actual internal name, any function that does not care for verification can proceed using that name.

5.4 Capability Operations

The following operations are provided by the kernel for handling capabilities:

1. **AlterRts**: This call allows the owner of an object to grant permission for others to access his object.

2. **ObtainCap**: Any object can issue this call to request a capability for a particular object. There are two parameters to this call. The first one specifies the object name (e.g. File1) and the second one is a capability that identifies the caller. Note that it is in this case that the caller might need to supply further location information if the system cannot resolve the name unambiguously.

3. **CopyCap**: To produce an identical copy of a given capability under the requirement that the given capability is duplicable (i.e. **CopyRts** is set).

4. **RemoveRts**: To remove a right from a capability. Except for **RestrictRts**, any rights can be removed from a capability.

---

because the internal name has a much larger range. Furthermore, one has to forge the *copy_no* as well. For example, if User A changes the internal name in his capability to 1001 which happens to be a valid *object_id*, then he has to guess a correct *copy_no* that is defined for object 1001.

Chapter 6

Context Switching and Scheduling

FLEX supports the concept of a process at the user level as an executable object. It also makes use of the process concept in implementing the concept of an object (i.e. the object coordinator). This chapter describes the context switching mechanism which is central to process management. It explains how the mechanism of context switching is separated from the policy of context switching in FLEX. The design of the dispatcher which lies within the Local Process Manager as well as the scheduler which lies outside the kernel are described. In the next section, we will briefly review the general ideas of the **process model** which is used in the design of traditional, process-oriented operating systems. More detail treatment of the model can be found in [Cal82,Com84,PS85,Tan87]. We will then present the treatment of processes in FLEX.

6.1 The Process Model

A process is a program in execution. The OS achieves the illusion of concurrent processing by rapidly switching the CPU among many processes. Such *context switchings* involve saving the state information of a process, selecting another process to run from those that are ready and giving the CPU to the newly selected process. Most OSs keep the information about a process in a data structure called the **process table**. For each process, there is one entry in the process table. This

entry contains information about the process state, the process identifier (pid), its program counter, its registers, memory allocation information and everything that will be destroyed when context switching is performed.

A process changes state as a program executes. Figure 6.1 shows a simple state transition diagram for a process. To execute a program, a process is created. The process enters the *ready* state when it has acquired the necessary resources and is ready to run. A process is in *running* state when it is given the CPU. A running process may move to the *blocked* state if it cannot execute because some event has been requested but not yet completed. Examples of such events are I/O requests, page faults, etc. When the event is completed, the process will move from the *blocked* state to the *ready* state.

A running process may also move to the *ready* state as a result of scheduling decisions. A process in the *ready* state moves back to the *running state* when the scheduler decides that it is time for it to run again. A running process eventually moves to the *exit* state when the program terminates.

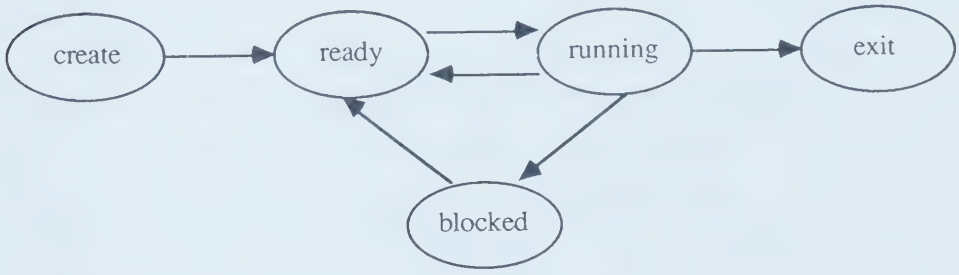

Figure 6.1: Process State Diagram

The scheduler in most operating systems can be divided into two parts: a long term scheduler (or job scheduler) and a short term scheduler (or CPU scheduler) [PS85]. The long term scheduler determines which jobs are admitted to the system for processing. These jobs might be spooled to some storage device and the long term scheduler's job is then to select and load them into memory for execution. On the other hand, the short term scheduler selects from among those jobs that

are in memory which are ready to run and gives the CPU to the selected one. The distinction between the two is that the long term scheduler executes much less frequently than the short term scheduler. The long term scheduler controls the number of processes in memory, therefore, it may need to be invoked only when a job enters or leaves the system. The short term scheduler, on the other hand, must select a new process quite often because a process may execute only a few milliseconds and then become blocked.

A large number of algorithms have been devised for the short term scheduler. Some of the more popular ones are given below:

1. **Round-robin**: Each process in turn is given a "time slice", at the end of which it is removed in favour of the next process that is ready to run. If the process becomes blocked or finished before the time elapses, the CPU switches to another process.

2. **Priority scheduling**: Each process is assigned a priority and the process with the highest priority is allowed to run. Priority can be assigned dynamically or statically.

3. **First come first serve (FCFS)**: Each process is serviced in the order of its arrival unless it becomes blocked. Some mechanism is built to prevent processes that run in an infinite loop to tie up the CPU.

4. **Shortest job first**: If the time it takes to run a job can be predicted, then *shortest job first* can be used to run the job that requires the smallest amount of time. This maximizes the throughput of the system.

It is often convenient to combine scheduling algorithms. For example, priority scheduling can be used together with FCFS: for those processes that have the same priority, FCFS can be used to break the tie.

Processes frequently need to communicate and coordinate with other processes. For example, the process managing the file system has to communicate with the process managing the disk to read or write a block. Sharing of resources like the

data buffer has to be coordinated. Semaphores are one of the simpliest process synchronization primitives and have been widely discussed in the literature. Communication between processes can be done in a number of ways, the most common ones are shared memory and message passing.

6.2 FLEX Design

In accordance with the principle of separating policies from mechanisms, the context switching mechanism is divided into two parts: the part that involves saving a process state information and giving control of the CPU to a selected process constitutes the mechanism. It is handled by the *dispatcher* that lies within the Local Process Manager inside the kernel. The part that selects another process to run from among those that are ready constitutes the policies. It is handled by the *scheduler* that runs as a user-level process.

Such a separation makes it possible to implement different scheduling algorithms. The scheduler running on one machine can implement a scheduling algorithm that is different from the one that runs on another machine. Alternatively, one can have multiple schedulers on a machine so that applications are free to choose the one that is more suitable for them. This would create an environment similar to that provided by VM on IBM machines [Dei84].

Using this scheme, the **process table** is kept inside the Local Process Manager. Each entry contains the state information for each process. However, each entry does not contain any scheduling information such as process priority or any scheduling parameters. The scheduling information is kept in the **ready queue** which is maintained by the scheduler.

Recall from the preceding chapter that an invocation of an operation within an object proceeds in three steps: message resolution, rights verification and operation activation. The last step involves the creation of a new process that is responsible for executing the invoked operation inside the object. The scheduler inserts this process into its **ready queue** according to its process priority. In FLEX, every newly

created process is given a priority which determines its eligibility for running. Some newly created processes might be assigned a higher priority than other processes – these processes are more favoured by the system. Examples of such processes are those that are invoked for operations within the kernel or the other OS services.

The scheduling algorithm employed by the current prototype uses a round robin with priority scheduling. Each process is allocated a certain time quantum, and if it exceeds its time quantum, it is preempted and feeds back into the priority queue. The priority of the process will also be decremented so that the newly created process will have a better chance to run than processes that exceed their time quantum.

Figure 6.2 shows an invocation scenario. One of the operations in object A is being invoked. After performing rights verification and message resolution, the OC for object A activates the operation by creating a new process and giving it to the scheduler to schedule for execution. The communication between object A and the scheduler is trapped by the IOCM. The scheduler communicates with the dispatcher to perform all context switchings.

The scheduler makes all the policy decisions. Based on the above scheduling algorithm, the scheduler is responsible for fixing the time quantum and adjusting process priorities. The scheduler also implements operations to maintain the **ready queue** according to process priority. The dispatcher, on the other hand, handles the low-level mechanism of context switching. This includes process suspension and process resumption. To suspend a process, the process state has to be changed, the registers and the program counter of the process have to be saved in its process table entry. To resume a process, the opposite such as restoring the registers has to be done. The dispatcher uses a mailbox to communicate with the scheduler. The scheduler fixes the time quantum and informs the dispatcher. The scheduler decides which process to run and deposits its identifier (pid) in the dispatcher's mailbox. The dispatcher executes processes in its mailbox in a first come first serve manner. When a process is blocked or exceeds the time quantum, its pid is returned to the scheduler to reschedule for execution.

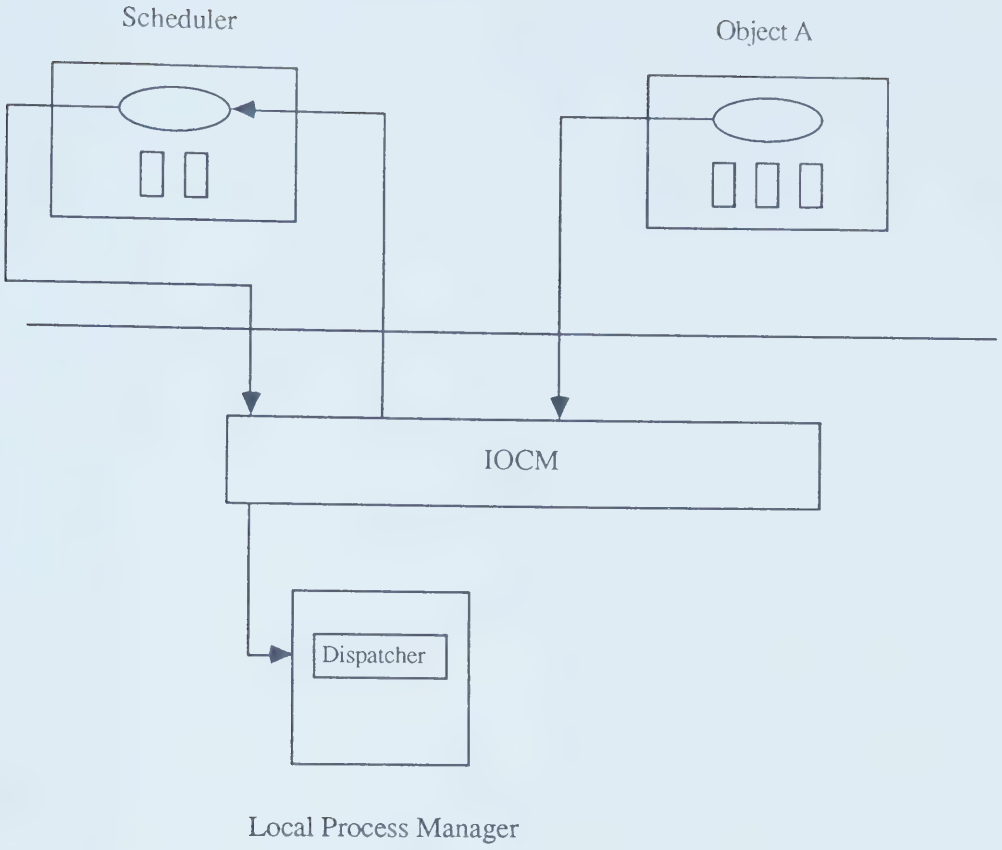

Figure 6.2: Object Invocation

6.3 Retrospective

The scheduling algorithm we employed is too simple for running a real operating system. To run the kernel on a bare machine, various adjustments will have to be made. For example, the kernel should run at the highest priority and should not be preempted. This is based on the assumption that the kernel components will work properly. They will block after finishing their work at which time user-level processes can be scheduled to run. We do not foresee too many problems in the conversion because the details have been described and well illustrated in literature [Com84,Tan87].

In section 6.1, we discussed long term versus short term scheduling. The long term scheduler is not included in the current prototype implementation. However it is not hard to visualize it as another user-level object. The long term scheduler will communicate with the memory manager to determine if there is enough memory to swap in processes from secondary storage for execution. Because the long term scheduler is executed much less frequently, implementing it at the user-level is clearly a right choice.

The current implementation demonstrates that it is possible to run the short term scheduler in user-space. Because the prototype does not run on a bare machine, we cannot collect statistics to evaluate if such an approach is justified in terms of performance. In our approach, to schedule a process for one time slice results in two messages: one from the scheduler to the kernel and one from the kernel to the scheduler. This can be a substantial overhead for a large process. For example, if the process requires 10 time slices, then 20 messages have to be sent. The messages can be saved if the short time scheduler is run inside the kernel. The problem, however, is that all applications are forced to use the algorithm provided in it. To remedy this, one might, perhaps, include various algorithms inside the kernel. Whenever the object coordinator creates a new process to execute an operation, the object coordinator can choose one of the algorithms. Nevertheless, providing buffering in the dispatcher (the mailbox in the current implementation) seems to be the right approach if the short term scheduler is to be moved out of the kernel. The buffering permits the dispatcher to quickly switch to the next available process in the mailbox whenever a process is finished or is blocked or exceeds its time quantum.

Chapter 7

Implementation

In this chapter, we describe the prototype implementation of the parts of FLEX which are discussed in this thesis. We discuss the computing environment and some of the tactics that we use to simulate the prototype.

7.1 Environment

The FLEX project was initiated around the beginning of 1987. Following the initial design, we decided to test out the feasibility of our design and our ideas before we embark on a full-scale implementation on one or more machines. Thus the previous year and a half was spent on implementing parts of the kernel under UNIX in a simulated environment. The programming language we used is C. The language gives us enough flexibility to simulate low level operations (e.g. bit manipulations) without having to use a low-level language (e.g. assembler). We did not attempt to modify the compiler at this stage.

The object manager, the protection mechanism and the dispatcher that are described in the preceding chapters are implemented as a UNIX process to run as the FLEX kernel. The size of this process is approximately 30kbytes. A UNIX process is also associated with each active user-level object (e.g. the scheduler). This process plays the role of the object coordinator.

7.2 Simulation Tactics

Since the prototype does not run on a bare machine, some tactics must be used to emulate the hardware. We relied heavily on the facilities provided by the UNIX operating system.

In Chapter 4, we described the object directory which consists of object descriptors. The object descriptors reside in secondary storage. We use files to emulate the secondary storage because files are the only permanent storage that is provided at the user-level in UNIX. File commands like **read**, **write**, **open**, **close** and **lseek** are used in the operations that handle the object directory.

We mentioned several times in this thesis that an object coordinator exists for every active object. To execute an object coordinator from the FLEX kernel, the UNIX command **execl** is used. Because the **execl** call will overlay the original program (i.e. the FLEX kernel) with the new one (i.e. the object coordinator), run that program and exit, we have to create a new process to invoke the **execl** call. The UNIX command **fork** is used to create a new process. The child process will execute the object coordinator and the parent process will execute the rest of the original program. The following pseudocode illustrates the situation.

```
if (fork() == 0 )
        execl ( objcoord, objcoord, parameters ) ;
else
        continue ;
```

In Chapter 6, we discussed context switching. Since the prototype runs on top of UNIX, the low level of context switching such as changing registers or suspending and resuming a process cannot be done. In order to test out the general algorithm, we again use the **fork** command to create a new process. The process identifier (pid) for each newly created process is stored in a simulated **process table** that is kept inside the Local Process Manager. To suspend and resume individual processes, the system call **kill** is used with the signal SIGSTOP(#17) and SIGCONT (#19).

The signal SIGSTOP will stop a process whereas the signal SIGCONT will resume a process from where it was stopped.

A real operating system uses the real-time clock to limit the amount of time a process can execute. For example, at each clock interrupt, the dispatcher will decrement a counter. When the counter reaches zero, the dispatcher preempts the running process. To simulate this situation, the system call **sleep** is used with the given amount of time. When the process that issues the **sleep** call wakes up, it is time to preempt the running process.

The rest of the implementation is quite straightforward and follows the design description given in the earlier chapters. In the next chapter, we will summarize the importance of this prototype and give a description of some of the ongoing research.

Chapter 8

Conclusion

In this thesis, we have argued for an architectural paradigm for integrating distributed database management systems with distributed operating systems. The three principles in the paradigm are:

- use of a client-server approach to structure OS modules
- separating policies from mechanisms
- use of an object-oriented design methodology

We presented the design of the distributed operating system FLEX that implements this paradigm. Several components in FLEX are described in detail and they include Object Management, Access Control and Protection as well as Scheduling.

Following the initial design, we decided to test the feasibility of our design before we embark on a full-scale implementation on one or more machines. As a result, all the parts described in this thesis were implemented in a simulated environment that runs under UNIX. The Object Manager, the Inter-Object Communication Manager and the Local Process Manager, for example, are linked together as a UNIX process that runs as the FLEX kernel. The scheduler is implemented as another UNIX process that communicates with the kernel via messages. Some user-level objects are written to run as drivers. They issue requests to invoke another object, to obtain a capability etc.

The initial prototyping proved to be very useful for the following reasons:

- It helped us to concentrate on the unique aspects of the FLEX kernel, without having to spend considerable time in rewriting device drivers or process dispatchers, which would have involved using a low-level language like assembler.

- It assisted us in validating (or invalidating) some of the design decisions. For example, we realize that it is important to provide uniform access to all objects via capabilities. In the initial design, system calls were used to access kernel objects. This proved to be inappropriate because essentially the kernel becomes a large domain and this contradicts the capability-based approach. Instead of system calls, capabilities should be used to permit access to kernel objects.

- It permitted us to delay making a commitment to a specific computer and language for implementation.

At the present, we believe that our approach is flexible enough for supporting a distributed database system which implements most of the policies in its own address space. However, this conjecture remains to be proven. The proof will take a number of years of research and may well result in a negative result. We believe that even this negative result is important because it will eliminate a significant architectural contender from further investigation.

8.1 Future Research

The FLEX project was started about one and a half years ago. The following are some of the ongoing and future research:

- Porting the initial prototype to a bare machine. The environment we will be working on is Sun Workstations (mostly Sun 3/50's).

- Developing a number of OS modules such as the file system, the user shell etc. to turn FLEX into a full-fledged operating system.

- Validating that the design is suitable for database applications.

- Refining the definition of an object so that it is semantically more sound and more general.
- Developing an object-oriented distributed database based on the FLEX kernel.
- Investigating the possibility of supporting transactions as an operating system service.
- Investigating the issues for supporting a heterogeneous distributed database management system on top of FLEX.

Bibliography

[ABLN85] G.T. Almes, A.P. Black, E.D. Lazowska, and J.D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, SE-11(1), January 1985.

[Bac86] M.J. Bach. *The Design of the Unix Operating System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Bla79] M. Blasgen. The convoy phenomenon. *Operating System Review*, 13(2):20–25, April 1979.

[Cal82] P. Calingaert. *Operating System Elements: A User Perspective.* Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[CD86] H.T. Chou and D.J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. *Algorithmica*, 1(3):311–336, 1986.

[Che84] D.R. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, April 1984.

[Cho85] H.T. Chou. *Buffer Management of Database Systems.* PhD thesis, Madison, Wisconsin: University of Wisconsin, 1985.

[CJ75] E. Cohen and D. Jefferson. Protection in the Hydra Operating System. In *Proceedings ACM Symposium on Operating Systems Principles*, pages 141–160, 1975.

[CM84] D.R. Cheriton and T.P. Mann. Uniform Access to Distributed Name Interpretation in the V-System. In *Proceedings of the 4th International Conference on Distributed Computer Systems*, 1984.

[Com84] D. Comer. *Operating System Design: The Xinu Approach.* Prentice-Hall, Toronto, Canada, 1984.

[CP84] J. Ceri and G. Pelagatti. *Distributed Databases Principles and Systems.* McGraw-Hill Computing Science Series, Toronto, Ontario, 1984.

[CZ85] D.R. Cheriton and W. Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems*, 3(2), May 1985.

[Dat84] C.J. Date. *Guide to DB2.* Addison-Wesley, Reading, Mass., 1984.

[DB83] P.J. Denning and R.L. Brown. Should Distributed Systems be Hidden? In *Int'l. Workshop on Computer System Organization*, pages 49–60, IEEE Computer Society Press, 1983.

[Dei84] H.M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley, Reading, Mass., 1984.

[DH66] J.B. Dennis and Van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3), March 1966.

[DLS85] P. Dasgupta, R.J. LeBlanc, and E. Spafford. *The Clouds Project: Design and Implementations of a Fault-Tolerant Distributed Operating System.* Technical Report Technical Report GIT-ICS-85/29, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985.

[DMN68] O.J. Dahl, B. Myhrhaung, and K. Nygaard. *The Simula 67 Common Base Language.* Norwegian Computing Centre, Oslo, Norway, 1968.

[Fab74] R.S. Fabry. Capability-Based Addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[Fle86] B.D. Fleisch. Distributed System V IPC in Locus: A Design and Implementation Retrospective. *ACM Sigcomm 86 Symposium Communications Architecture and Protocols*, 16(3), August 1986.

[GC86] E. Gehringer and R. Colwell. Fast Object-Oriented Procedure Calls: Lessons from the Intel 432. In *13th Annual International Symposium on Computer Architecture*, pages 92–101, 1986.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Mass., 1983.

[Gra78] J.N. Gray. Notes on Data Base Operating Systems. In Goos and Hartmanis, editors, *Operating Systems - An Advanced Course*, Springer-Verlag, 1978.

[Hol83] R.C. Holt. *Concurrent Euclid, the Unix System, and Tunis.* Addison-Wesley, Reading, Mass., 1983.

[Ins85] *Institute of Electrical and Electronics Engineers.* ANSI/IEEE Standard 802.2 Logical Link Control, 1985.

[Int83] *International Standards Organization.* Information Processing Systems - Open Systems Interconnection - Basic Reference Model, 1983.

[Jon78] A.K. Jones. The Object Model: A Conceptual Tool For Structuring Software. In Goos and Hartmanis, editors, *Operating Systems - An Advanced Course*, Springer-Verlag, 1978.

[JR86] M.B. Jones and R.F. Rashid. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. In *OOPSLA 86 Conference Proceedings*, 1986.

[JW75] A.K. Jones and W.A. Wulf. Towards the Design of Secure Systems. *Software–Practice and Experience*, 5(4):321–336, October-December 1975.

[LCC*75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/Mechanism Separation in Hydra. In *Proc. ACM Symposium on Operating Systems Principles*, pages 132–140, 1975.

[Lev84] H.M. Levy. *Capability-Based Computer Systems.* Digital Equipment Corporation, Bedford, Mass., 1984.

[Li88] Y. Li. *Inter-Object Communication in FLEX.* Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1988.

[Lin76] T. Linden. Operating System Structures to Support Security and Reliable Software. *Computing Surveys*, 8(4):309–445, December 1976.

[LLA*81] E.D. Lazowska, H.M. Levy, G.T. Almes, H.J. Fischer, R.J. Fowler, and S.C. Vestal. The Architecture of the Eden system. In *Proc. 8th Symp. Operating Systems Principles*, pages 148–159, December 1981.

[MT84] S.J. Mullender and A.S. Tanenbaum. Protection and Resource Control in Distributed Operating Systems. *Computer Networks*, 8:421–432, 1984.

[MT86] S.J. Mullender and A.S. Tanenbaum. The Design of a Capability-Based Distributed Operating System. *Computer Journal*, 289–299, August 1986.

[Org83] E.I. Organick. *A Programmer's View of the Intel 432 System.* McGraw-Hill Book Company, New York, N.Y, 1983.

[Ozs88] M.T. Ozsu. *Distributed Database Operating Systems.* Technical Report TR88-2, Department of Computing Science, University of Alberta, Edmonton, Canada, February 1988.

[PD87] D.V. Pitts and P. Dasgupta. *Object Memory and Storage Management in the Clouds Kernel.* Technical Report Technical Report GIT-ICS-87/15, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.

[Pet78] J.L. Peterson. Notes on a Workshop on Distributed Computing. In *Proc. ACM Symposium on Operating Systems Principles*, pages 18–27, 1978.

[PS85] J.L. Peterson and A. Silberschatz. *Operating System Concepts.* Addision Wesley, Don Mills, Ontario, second edition edition, 1985.

[PW85] G.J. Popek and B.J. Walker. *The Locus Distributed System Architecture.* The MIT Press, Cambridge, Mass, 1985.

[QSP85] J.S. Quarterman, A. Silberschatz, and J.L. Peterson. 4.2 BSD and 4.3 BSD as Examples of the Unix System. *Computing Surveys*, 17(4), December 1985.

[Ras86] R.F. Rashid. Threads of a New System. *Unix Review*, 4(8), August 1986.

[SGK*85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Usenix Conference and Exhibition Proceedings*, Summer 1985.

[SKWH76] M. Stonebraker, P. Kreps, W. Wong, and G. Held. The Design and Implementation of INGRES. *ACM Trans. Database Systems*, 1(3):198–222, September 1976.

[SS75] J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, Septermber 1975.

[SS81] D. Skeen and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. In *Proc. Fifth Berkeley Workshop on Distributed Data Man. and Comp. Networks*, pages 129–142, 1981.

[SS86] G.M. Sacco and M. Schkolnick. Buffer Management in Relational Database Systems. *ACM Trans. Datebase Systems*, 11(4):473–498, December 1986.

[Sta86] W. Stallings. A Tutorial on the IEEE 802 Local Area Network Standard. In R.L. Pickholtz, editor, *Local Area & Multiple Access Networks*, Computer Science Press, 1986.

[Sto81] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7), July 1981.

[Tan87] A.S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[Tay87] R. Taylor. Data Server Architectures: Experiences and Lessons. In *Proc. CIPS Edmonton Conference Edmonton, Canada*, pages 334–342, November 1987.

[Tho78] K. Thompson. Unix Implementation. *Bell System Tech. Journal*, 57(6, Pt,2):1931–1946, 1978.

[TM82] A.S. Tanenbaum and S.J. Mullender. Operating System Requirements for Distributed Data Base Systems. In Schneider, editor, *Distributed Data Bases*, pages 105–114, North-Holland, Amsterdam, 1982.

[TMvR86] A.S. Tanenbaum, S.J. Mullender, and R. van Renesse. Using Sparse Capabilities in a Distributed Operating System. In *IEEE Distributed Computing Systems*, 1986.

[TvR85] A.S. Tanenbaum and R. van Renesse. Distributed Operating Systems. *Computing Surveys*, 17(4):419–470, December 1985.

[WCC*74] W.A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–345, June 1974.

[WLH81] W.A. Wulf, R. Levin, and S. Harbison. *Hydra/C.mmp : An Experimental Computer System.* McGraw Hill, New York, N.Y, 1981.

B46040